Freescale Semiconductor, Inc.

Suite56™ DSP Simulator

User's Manual, Release 6.3

DSPS56SIMUM/D
Document Rev. 3.0, 07/1999

MOTOROLA

Freescale Semiconductor, Inc.

Suite56, OnCe, and MFAX are trademarks of Motorola, Inc.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.

Motorola and are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

All other tradenames, trademarks, and registered trademarks are the property of their respective owners.

© Copyright Motorola, Inc., 1999. All rights reserved.

Table of Contents

Chapter 1
Introduction & Getting Started

1.1 Introduction to Motorola's Suite56™ DSP Simulator 1-1
1.2 DSP Simulator Features 1-2
1.3 Entering Commands 1-2
1.4 Getting Started with the DSP Simulator 1-3
1.5 Setting and Clearing the Path 1-4
1.6 Loading Object Files 1-6
1.7 Setting Up the Display Environment 1-7
1.8 Using a Watch List 1-9
1.9 Setting and Modifying Breakpoints 1-10
1.10 Starting the Execution of Instructions with GO 1-13

Chapter 2
Controlling Execution

2.1 Starting the Execution of Instructions with GO 2-1
2.2 STEP Through Instructions 2-1
2.3 TRACE the Execution of Instructions 2-2
2.4 Executing the NEXT Instruction 2-3
2.5 Providing an UNTIL Condition 2-4
2.6 Pausing Execution with WAIT 2-5
2.7 Allowing the Current Function to FINISH
after an UNTIL Condition or a Breakpoint 2-6
2.8 Stopping Program Execution 2-7
2.9 Resetting the Device 2-7

Chapter 3
Object Files and Data Files

3.1 Displaying the Current PATH 3-1
3.2 Saving Object Files 3-3

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 4
Managing Memory and Registers

4.1 Displaying Register Values 4-1
4.2 Changing Register Values 4-3
4.3 Displaying Memory Values 4-3
4.4 Changing Memory Values 4-5
4.5 Copying Memory from One Block to Another 4-6
4.6 Disassembling Code Stored in Memory 4-7
4.7 Displaying a History of Recent Instructions 4-8

Chapter 5
Device I/O and Peripheral Simulation

5.1 Introduction to Device I/O (Input/Output) 5-1
5.2 How Are I/O Files Formatted? 5-1
5.3 Repeat Punctuation 5-2
5.4 Comments 5-2
5.5 Timing Information 5-3
5.6 Peripheral Data 5-3
5.7 Port Data 5-4
5.8 Memory Data 5-5
5.9 Pin or Pin Group Data 5-6
5.10 Terminal Input of Data Values 5-7
5.11 Assigning an Input File 5-8
5.12 Assigning an Output File 5-11

Chapter 6
Simulator and Device Configurations

6.1 Introduction to Simulator Configuration 6-1
6.2 Setting the Default DEVICE 6-2
6.3 Changing the Radix 6-2
6.4 Loading Simulator State Files 6-4
6.5 Changing and Saving Window Preferences 6-6

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 7
Debugging C Source Code

7.1 Introduction to Debugging C Source Code 7-1
7.2 Displaying the Call Stack 7-1
7.3 Moving Up and Down the Call Stack 7-2
7.4 Dynamically Displaying C Function Calls 7-4
7.5 Enabling/Disabling I/O Streams 7-5
7.6 Redirecting an I/O Stream 7-5
7.7 Display the Type of a C Variable or Expression 7-6

Chapter 8
Macros, Scripts and Log Files

8.1 Creating and Running a Command Macro 8-1
8.2 Logging Output from the Session Window 8-2

Chapter 9
Expressions

9.1 Introduction to Expressions 9-1
9.2 Evaluate Expressions 9-1
9.3 Using Memory Space Symbols 9-2
9.4 Using Register Name Symbols 9-2
9.5 Using Assembler Debug Symbols 9-3
9.6 Using Constants 9-4
9.7 Numeric Constants 9-4
9.8 Operators in Expressions 9-5
9.9 Operator Precedence 9-7
9.10 Setting Up and Modifying a Watch List 9-8

Chapter 10
Simulator Tool Bar

10.1 Using the Tool Bar 10-1
10.2 Go Button 10-2
10.3 Stop Button 10-2
10.4 Step Button 10-3
10.5 Next Button 10-4
10.6 Finish Button 10-5
10.7 Device Button 10-6
10.8 Repeat Button 10-7
10.9 Reset Button 10-8

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 11
Displaying Information

11.1 Display the Current Breakpoints 11-1
11.2 Displaying the Radix 11-2
11.3 Command Window 11-2
11.4 Session Window 11-4
11.5 Assembly Window. 11-5
11.6 Source Window 11-6
11.7 Stack Window 11-7

Chapter 12
Command Reference

12.1 Command Overview 12-1
12.2 Command Syntax. 12-4
12.3 Command Parameters: List of Abbreviations 12-5
12.4 asm - Single Line Interactive Assembler. 12-7
12.5 break - Set, Modify, or Clear Breakpoint 12-8
12.6 change - Change Register or Memory Value 12-13
12.7 copy - Copy a Memory Block 12-14
12.8 device - Multiple Device Simulation. 12-15
12.9 disassemble - Object Code Disassembler 12-16
12.10 display - Display Register or Memory. 12-17
12.11 down - Move Down the C Function Call Stack. 12-19
12.12 evaluate - Evaluate an Expression 12-20
12.13 finish - Execute Until End of Current Subroutine 12-21
12.14 frame - Select C Function Call Stack Frame 12-21
12.15 go - Execute DSP Program 12-22
12.16 help - Simulator Help Text 12-23
12.17 history -Disassemble Previously Executed Instruction 12-24
12.18 input - Assign Input File 12-24
12.19 list - List Source File Lines 12-27
12.20 load - Load DSP Files or Configuration 12-28
12.21 log - Log Commands, Session, Profile 12-30
12.22 more - Enable/Disable Session Paging Control. 12-31
12.23 next- Step Over Subroutine Calls or Macros. 12-32
12.24 output - Assign Output File 12-33
12.25 path - Specify Default Pathname 12-35
12.26 quit - Quit Simulator Session. 12-36
12.27 radix - Change Input or Display Radix 12-36

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.28 redirect - Redirect stdin/stdout/stderr for C Programs 12-38
12.29 reset - Reset Device or State 12-39
12.30 save - Save Simulator File 12-40
12.31 step - Step Through DSP Program. 12-41
12.32 streams - Enable/Disable Handling of I/O for C Programs 12-42
12.33 system - Execute System Command 12-43
12.34 trace - Trace Through DSP Program 12-44
12.35 type - Display the Result Type of C Expression 12-45
12.36 unlock - Unlock Password Protected Device Type 12-46
12.37 until - Execute Until Address 12-46
12.38 up - Move Up the C Function Call Stack 12-47
12.39 view - Select Display Mode. 12-47
12.40 wait - Wait Specified Time 12-48
12.41 wasm - GUI Assembly window. 12-48
12.42 watch - Set, Modify, View, or Clear Watch item 12-49
12.43 wbreakpoint - GUI Breakpoint Window 12-50
12.44 wcalls - GUI C Calls Stack window 12-50
12.45 wcommand - GUI Command window. 12-51
12.46 where - GUI C Calls Stack window 12-51
12.47 winput - GUI File Input window 12-52
12.48 wlist - GUI list window 12-52
12.49 wmemory - GUI Memory window 12-53
12.50 woutput - GUI File Output window. 12-54
12.51 wregister - GUI Register window 12-54
12.52 wsession - GUI Session window 12-55
12.53 wsource - GUI Source window 12-55
12.54 wstack - GUI Stack window 12-56
12.55 wwatch - GUI Watch window 12-56

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 13
C Library Functions

13.1 C Object Libraries 13-2
13.2 Simulator Object Library Entry Points 13-2
13.3 Simulator External Memory Functions 13-21
13.4 Simulator Screen Management Functions 13-26
13.5 Non-Display Simulator 13-31
13.6 Multiple Device Simulation. 13-33
13.7 Reserved Function Names 13-36
13.8 Simulator Global Variables 13-36
13.9 Modification of Simulator Global Structures 13-36
13.10 Modification of Device Global Structures. 13-37

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

List of Tables

1-1	List of Operators	1-11
1-2	Breakpoint Actions	1-12
5-1	Repeat Punctuation Input Data	5-2
5-2	Input Memory Data	5-6
5-3	Pin or Pin Group Input Data	5-7
5-4	Terminal Input Data	5-8
9-1	Valid Forms of Symbol Names and Line Numbers	9-3
12-1	Command Syntax Elements in Motorola DSP Documentation	12-4
12-2	Command Parameter Abbreviations	12-5
12-3	asm Commands	12-7
12-4	Breakpoint Operators	12-9
12-6	Break_Action Parameters	12-10
12-5	Flag Variables in a Breakpoint Expression	12-10
12-7	BREAK Commands	12-12
12-8	CHANGE Commands	12-13
12-9	COPY Commands	12-14
12-10	DEVICE Command	12-15
12-11	DISASSEMBLE Commands	12-16
12-12	DISPLAY Enable Keywords	12-17
12-13	DISPLAY Commands	12-18
12-14	DISPLAY Commands Without Enable Keywords	12-18
12-15	DOWN Commands	12-19
12-16	EVALUATE Commands	12-20
12-17	FRAME Commands	12-21
12-18	GO Commands	12-22
12-19	HELP Syntax	12-23
12-20	List of Help Topic Keywords	12-23

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12-21 INPUT Commands 12-25
12-22 LIST Commands. 12-27
12-23 LOAD Commands 12-29
12-24 LOG Commands. 12-31
12-25 MORE Commands 12-31
12-26 NEXT Commands 12-32
12-27 OUTPUT Commands 12-34
12-28 PATH Commands. 12-35
12-29 QUIT Commands 12-36
12-30 RADIX Commands 12-37
12-31 REDIRECT Commands 12-38
12-32 RESET Commands. 12-39
12-33 SAVE Commands. 12-40
12-34 STEP Commands 12-41
12-35 STREAMS Commands. 12-42
12-36 SYSTEM Commands 12-43
12-37 TRACE Commands 12-44
12-38 TYPE Commands 12-45
12-40 UNTIL Commands 12-46
12-39 UNLOCK Commands 12-46
12-42 VIEW Commands. 12-47
12-41 UP Commands 12-47
12-43 WASM Commands. 12-48
12-44 WATCH commands 12-49
12-45 WBREAKPOINT Commands 12-50
12-46 WCALLS Commands. 12-50
12-47 WCOMMAND Commands 12-51
12-48 WHERE Commands. 12-51
12-50 WLIST Commands. 12-52
12-49 WINPUT Commands 12-52
12-51 WMEMORY Command. 12-53

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12-52 WOUTPUT Commands 12-54
12-53 WREGISTER Commands 12-54
12-54 WSESSION Commands 12-55
12-55 WSOURCE Commands 12-55
12-56 WSTACK Commands 12-56
12-57 WWATCH commands 12-57
13-1 Higher Level Functions 13-3
13-2 Integer Code 13-4
13-3 External Memory Functions 13-21
13-4 Screen Management Functions 13-27
13-5 Lower Level Structures & Define Constants 13-38

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

List of Figures

1-1 DSP Simulator Command Window 1-2
1-2 Setting the Working Directory Path 1-5
1-3 Setting Up the Display Environment 1-8
1-4 Add Item to a Watch List 1-9
1-5 Execution Instructions with Go 1-14
2-1 Step through Program Instructions 2-2
2-2 Trace Program Execution 2-3
2-3 Providing an Until Condition 2-5
2-4 Resetting the Device Registers 2-8
3-1 Displaying Current Paths 3-2
4-1 DSP Simulator Register Dialog Box. 4-1
4-2 Displaying the Register Values. 4-2
4-3 Changing the Value of Register 4-3
4-4 DSP Simulator Memory Dialog Box 4-4
4-5 Displaying Memory Values 4-4
4-6 Changing the Value in Memory 4-5
4-7 Copying Memory from One Block to Another. 4-6
4-8 Disassembling Code Stored in Memory 4-7
5-1 Providing Input Data. 5-9
5-2 Creating an Output File. 5-11
6-1 Set Default Device Dialog Box. 6-2
6-2 Set Default Radix Dialog Box 6-3
6-3 Set Register or Memory Radix Dialog Box 6-4
6-4 Window Preferences Dialog Box 6-6
7-1 Displaying the Call Stack 7-1
7-2 Moving Up the Call Stack. 7-2
7-3 Moving Down the Call Stack 7-3

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

7-4 Dynamic Display of C Function Calls 7-4
7-5 Displaying the Type of a C Variable or Expression 7-6
9-1 Evaluating an Expression 9-1
9-2 Displaying a Value in a Watch List 9-8
10-1 Go Button 10-2
10-2 Stop Button 10-2
10-3 Step Button 10-3
10-4 Next Button. 10-4
10-5 Finish Button 10-5
10-6 Device Button 10-6
10-7 Repeat Button 10-7
10-8 Reset Button 10-8
11-1 Displaying the Current Breakpoints 11-1
11-2 Displaying the Command Window 11-3
11-3 Pausing Output to the Session Window 11-5
11-4 Using the Assembly Window 11-5
11-5 Displaying the Source Window 11-6

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

List of Examples

5-1 Comment Code 5-2
13-1 dspt_masm_xxxxx 13-5
13-2 dspt_unasm_xxxxx 13-5
13-3 dsp_exec 13-6
13-4 dsp_findmem. 13-6
13-5 dsp_findpin 13-7
13-6 dsp_findport 13-7
13-7 dsp_findrg 13-8
13-8 dsp_free. 13-8
13-9 dsp_fmem 13-9
13-10 dsp_init 13-10
13-11 dsp_Idmem 13-10
13-12 dsp_load 13-11
13-13 dsp_new 13-11
13-14 dsp_path 13-11
13-15 dsp_rapin. 13-12
13-16 dsp_rmem 13-13
13-17 dsp_rpin. 13-13
13-18 dsp_rport 13-14
13-19 dsp_rreg. 13-14
13-20 dsp_save 13-15
13-21 dsp_startup 13-15
13-22 dsp_unlock 13-16
13-23 dsp_wapin 13-16
13-24 dsp_wmem 13-17
13-25 dsp_wpin 13-17
13-26 dsp_wport 13-18
13-27 dsp_wreg 13-19
13-28 sim_docmd 13-19

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

13-29 sim_gmcmd 13-20
13-30 sim_gtcmd 13-21
13-31 dsp_alloc 13-22
13-32 dspl_xmfree 13-23
13-33 dspl_xminit 13-23
13-34 dspl_xmload 13-24
13-35 dspl_xmnew 13-24
13-36 dsp_xmrd 13-25
13-37 dspl_xmsave 13-25
13-38 dspl_xmwr 13-26
13-39 Code to Create New Device 13-32

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 1
Introduction & Getting Started

1.1 Introduction to Motorola's Suite56™ DSP Simulator

The DSP Simulator is a software tool for developing programs and algorithms for Motorola Digital Signal Processors (DSPs). The DSP Simulator duplicates the functions of supported Motorola DSP chips:

- all on-chip peripheral operations
- all memory and register updates associated with program code execution
- all exception processing activity

The Simulator also simulates the device's pipelined bus activity, enabling the Simulator to provide you, the developer, an accurate measurement of code execution time—critical in DSP applications.

The Simulator executes object code generated by the Suite56 DSP Assembler (see the *Suite56 DSP Assembler Reference Manual*) or the Simulator's internal single-line assembler. The Simulator also executes object code generated by C compilers, such as Motorola's Star*Core 100 Family C/C++ Compiler, or Suite56 compilers for the 56000, 56300, 56600 and 56800 families. The object code is loaded into the simulated device's memory map. The entire internal and external memory space of the DSP is simulated. During program debugging you can display and change any of the device's registers or memory locations.

Instruction execution can proceed until the program encounters a user-defined breakpoint, or, in single-step mode, until it reaches a stopping point by executing a specified number of instructions or cycles.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

1.2 DSP Simulator Features

Features of the Motorola DSP Simulator include the following:

- Multiple device simulation
- Source level symbolic debug of assembly and C source programs
- Conditional or unconditional breakpoints
- Program patching using a single-line assembler/disassembler
- Instruction and cycle timing counters
- Session and/or command logging for later reference
- Input/output from/to ASCII files for device peripherals
- Help file and help line display of Simulator commands
- Command macros (editable scripts for executing multiple Simulator commands)
- Display enable/disable of registers and memory
- Fractional/hexadecimal/decimal/binary calculator

1.3 Entering Commands

You will perform most of your work with the Simulator using the menu bar and the toolbar. But, sometimes it is convenient to perform some commands from a command line, and Motorola's DSP Simulator provides you the option of entering commands at a command line. The command line is a part of the Command window, shown in Figure 1-1.

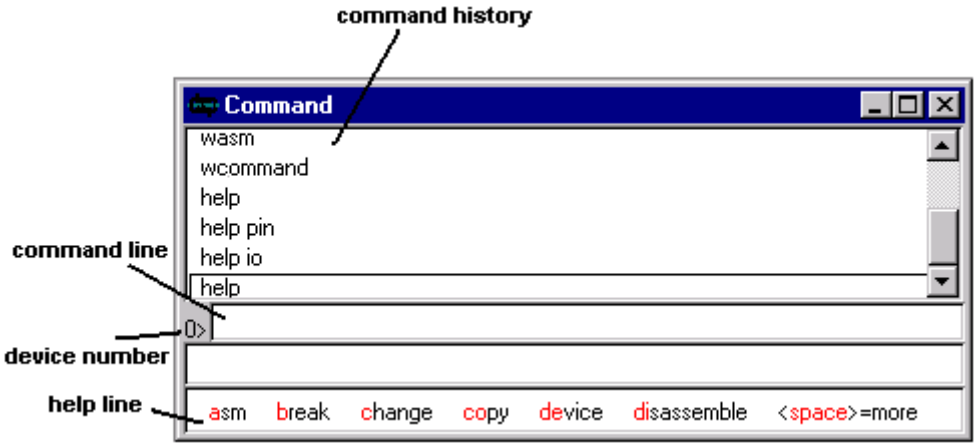

Figure 1-1. DSP Simulator Command Window

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

The Command window displays several common commands on the help line. You can display the remaining commands by pressing the SPACE bar when the cursor is at the beginning of the command line.

You do not have to type the complete command. The Simulator recognizes each command when you type the first one to three characters of the command. The help line highlights the minimum number of required characters for each command. The help line displays the syntax for a particular command when you type the required characters and then pressing the SPACE bar.

The command line considers any text that follows a semicolon a user comment. You might use comments to log output from the Session window or to create and run a command macro.

The command line executes the command when you press the ENTER key or the CARRIAGE RETURN key. If you enter a command that is not a valid Simulator command, the Simulator interprets the command as the name of a command macro and executes the macro, if it exists.

For more information about command syntax, see Section 12.1, "Command Overview," Section 12.2, "Command Syntax," and Section 12.3, "Command Parameters: List of Abbreviations."

1.4 Getting Started with the DSP Simulator

The Motorola Suite 56 DSP Simulator gives you flexibility in debugging your object code. You can take multiple approaches; there is no one “correct” way in which to go about debugging your code. Whichever approach you choose, you will want to become familiar with some fairly common windows and commands. Here are a few steps to help you begin a typical session with the Simulator.

Typically, you will want to:

1. Set the path of the working directory.
2. Load an object file.
3. Set up the display environment: open windows to view the source code, assembler code, register values, and memory values.
4. Use a watch list.
5. Set and modify break points.
6. Go or step through instructions.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

When you become familiar with the Simulator, you will not have to manually perform each step described above. You can position your windows as you like them by saving the windows settings using **Window Preferences**. Command macros can set the path, load the program, and set up watch expressions.

Once you are comfortable with these basics, you can proceed to more complex debugging using simulated device input and output (I/O). If you are using C code as your source code, you use specific commands useful in debugging C source code.

Refer to the following sections of this manual to learn about more advanced techniques:

- Section 5.1, "Introduction to Device I/O (Input/Output)," on page 5-1
- Section 6.1, "Introduction to Simulator Configuration," on page 6-1
- Section 7.1, "Introduction to Debugging C Source Code," on page 7-1
- Section 9.1, "Introduction to Expressions," on page 9-1
- Section 10.1, "Using the Tool Bar," on page 10-1
- Section 12.1, "Command Overview," on page 12-1

1.5 Setting and Clearing the Path

The Suite 56 DSP Simulator uses two types of paths to save and access files:

- the working directory path
- alternate directory paths

The working directory is the primary path. The Simulator uses the working directory as the default when searching for an input file (assuming no path is explicitly specified with the input filename). The Simulator automatically searches alternate paths if it does not find the requested input file in the working directory. This allows you to keep object files, source files, and command macros in separate directories, but still access them without constantly redefining the path.

To set the path of the working directory:

1. From the **File** menu choose **Path** then select **Set**. A dialog box similar to the following appears:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

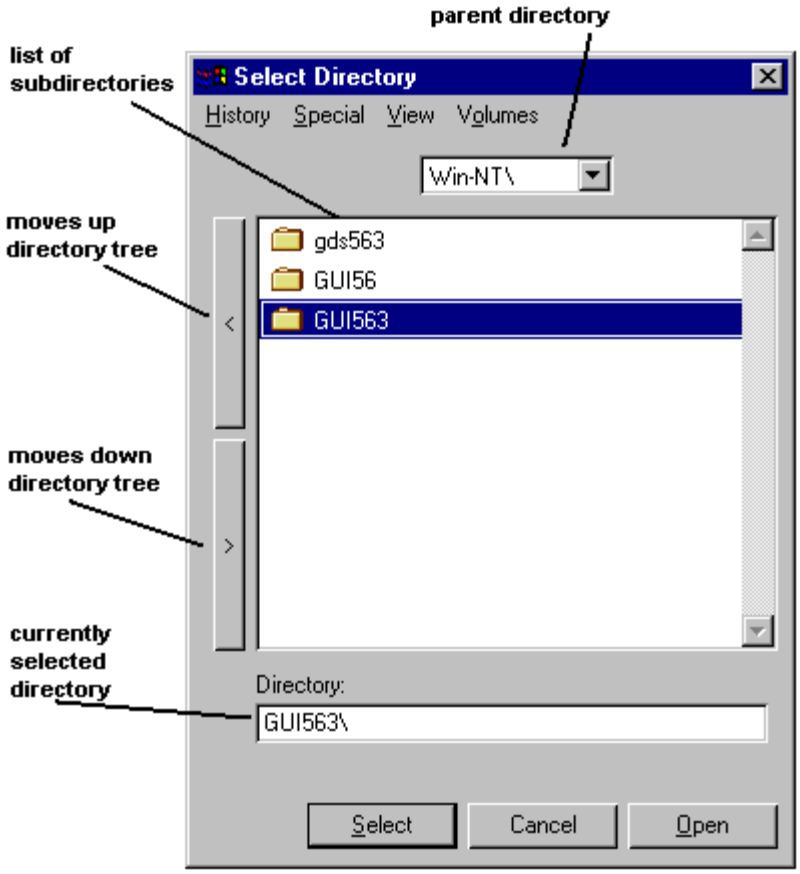

Figure 1-2. Setting the Working Directory Path

2. If appropriate, select another drive from the **Volumes** menu.
3. Move up or down the directory tree until the directory you want is in the list of subdirectories. There are a number of ways to do this. You can:
   - — move up and down the directory tree with the left arrow and right arrow buttons;
   - — move down the tree by double clicking in the list of subdirectories;
   - — move up the tree by selecting a directory from the drop down box that displays the parent directory;
   - — select a directory from the **History** menu, which shows recently selected directories.
4. Click once on the desired directory from the list of subdirectories so that it is highlighted.Notice that the highlighted directory appears in the area labeled Directory.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

5. Click on **Select**.The selected directory is now the working directory. **Display the Current PATH** to see the absolute path of the working directory

To set an alternate path:

1. From the **File** menu choose **Path** then select **Add**.
2. If appropriate, select another drive from the **Volumes** menu.
3. Move up or down the directory tree until the directory you want is in the list of subdirectories.
4. Highlight the desired directory by single clicking on it in the list of subdirectories.
5. Click on **Select**. The selected directory is now added to the end of the list of alternate paths. **Display the Current PATH** to see the list of alternate paths.

To clear the alternate paths:

1. From the **File** menu, choose **Path** then select **Clear Alternate Path List.**The list of alternate paths is cleared for all devices. The path of the working directory is not cleared.

The Simulator maintains a separate working directory for each device. However, all devices share the same alternate directory paths. To be safe, always check the path of the working directory after adding a device or setting the default device.

Remember that in order to change the path of a device that is not currently the default device, you must first change the default device, discussed in Section 6.2, "Setting the Default DEVICE," and Section 12.25, "path - Specify Default Pathname," on page 12-35. in Chapter 12, "Command Reference."

1.6 Loading Object Files

You can load DSP Object Module Format (OMF) files or DSP Common Object File Format (COFF) files directly into the Simulator memory. OMF files are identified by the .lod extension. COFF files are identified by the .cld extension. Motorola’s Suite56 DSP Assembler generates the OMF file format. To generate COFF files with the Suite56 Assembler, you can use the cldlod utility to generate .lod files; the cldlod utility converts files from COFF file format (.cld) to OMF file format (.lod). You can also generate both of the file formats with the DSP Simulator when you save object files.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

To load a COFF (.cld) file:

1. From the **File** menu, choose **Load** then select **Memory COFF**.
2. Under **Load** select **memory**, **debug symbols** or both.The Simulator will process the symbol and line number information contained in a COFF format object file (.cld file) only if the file was compiled or assembled with debugging enabled. (In the Motorola DSP Assembler this is represented by the `-g` option.) If symbol information has been loaded, the evaluator will accept symbol names or source file line numbers and translate them into an associated memory address.
3. Under **Filename** specify the filename of the object file to load or click on the **File** button to browse for the file.
4. Click **OK**. If the .cld file is not found in the selected directory, the Simulator will try to load the file from the working directory. If the file does not exist in the working directory, the Simulator will try to load the file from the alternate directories. An error message is displayed if the file is not found in any of these directories.

To load an OMF (.lod) file:

1. From the **File** menu, choose **Load** then select **Memory OMF**
2. Specify the filename in the dialog box. It is not necessary to type the extension with the filename. The Simulator assumes the .lod extension.
3. Click **Open**.

Keep in mind that the object file is loaded into the memory of the current device. If you want to load the file into another device, you must first select that device as the default device.

Chapter 6, "Simulator and Device Configurations" provides more details and **Chapter 12, "Command Reference"** provides a full command description.

1.7 Setting Up the Display Environment

The Simulator displays two windows when it first starts: the Session window and the Command window. You might need to scroll around to get both windows into view at the same time, but the screen should look something like this:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

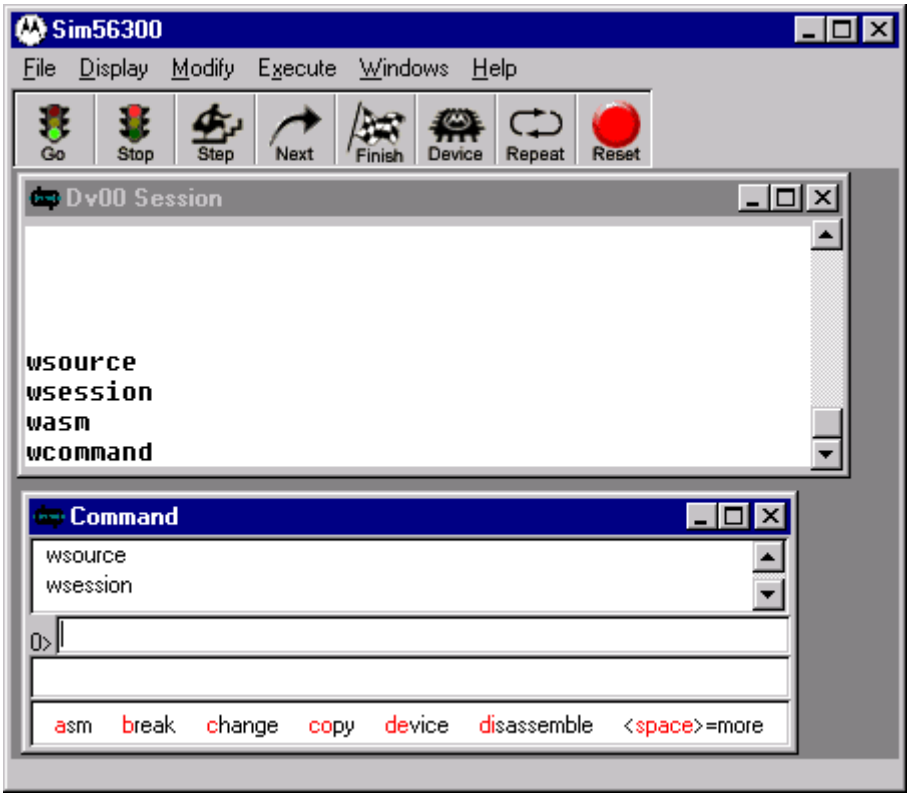

Figure 1-3. Setting Up the Display Environment

When we talk about the display environment we are simply talking about the way you arrange the windows of the Simulator on your monitor. There is no right or wrong way to set up the display environment. However, there are some windows that you will commonly want to have opened besides the Session and Command windows. These include:

- the Assembly window
- the Source window (if a COFF file with debugging information is loaded)
- a window displaying register values
- a window displaying memory values, and perhaps
- a WATCH List

If you are debugging a program written in C, keep the Call Stack window open so that you can see the call stack.

Depending on the size and resolution of your monitor, having all of these windows open at once will mean that there will be some overlapping. Your own experience with the Simulator will lead you to the best configuration of these windows

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

1.8 Using a Watch List

You can watch the contents of a specific memory location, register, or expression by setting up a Watch list. The Watch list is updated every time program execution is stopped.

The expression that you watch can be valid even if it is not calculated during program execution. You can use C expressions, but you must enclose them in curly brackets: {*c_expression*}. You can use symbolic references if you have loaded symbols from the object module.

To add an item to a Watch list:

1. From the **Windows** menu choose **Watch**. The following dialog box appears:

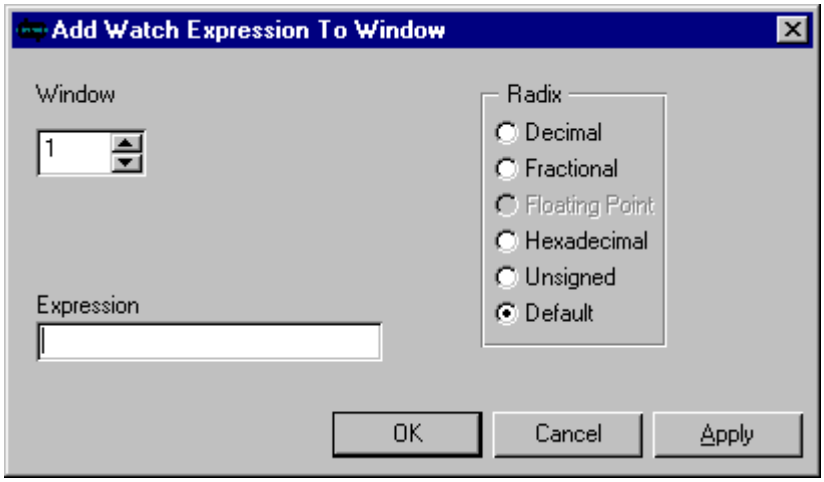

Figure 1-4. Add Item to a Watch List

2. Under **Window** select the window number that you want to assign to the Watch window. This is useful when you have more than one Watch window open.
3. Under **Expression**, type the expression that you want to appear in the Watch window. If the expression is a C expression, enclose it in curly brackets: {*c_expression*}.
4. Under **Radix**, select the radix format in which you want the variables displayed.
5. Click **OK**. The expression that you specified now appears in a Watch window: If the expression you type is not valid, you get an error message explaining why the expression is not valid.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Keep in mind that a C expression that refers to C variables can only be evaluated in the context in which the watch is established. That is, the variables in the expression are only valid while they are in scope. If one of the variables in an expression goes out of scope (because of either a function call or a return from a function), the value is replaced with the message Expression out of scope. When all elements of the expression are back in scope, the Watch window displays the value again.

An expression that has gone out of scope because of a function call can be evaluated and displayed by selecting the stack frame for the evaluation context. The stack frame assignment remains in effect only until the next instruction is executed. An expression that is out of scope because of a function exit cannot be evaluated until the function is again invoked since the expression's variables no longer exist.

Refer to the following sections of this manual to learn about more advanced topics: Section 7.3, "Moving Up and Down the Call Stack," on page 7-2; Section 9.1, "Introduction to Expressions," on page 9-1; and Section 11.2, "Displaying the Radix." Section 12.42, "watch - Set, Modify, View, or Clear Watch item," on page 12-49, provides a complete command description.

1.9 Setting and Modifying Breakpoints

You can use the Assembly window, (which is updated at each break in program execution), and the Source window, (which displays the source code loaded into the Simulator's memory), to set halt breakpoints. To set a halt breakpoint in the Assembly window, double click on an address or label field. To set or clear a halt breakpoint in the Source window, double click on any source code line . The Assembly window and the Source window indicate the breakpoints you have set. Enabled breakpoints appear in blue. Disabled breakpoints appear in pink. Chapter 11, "Displaying Information" provides a detailed discussion of the Simulator's display capabilities, including the Assembly and Source windows.

To set a breakpoint with conditions, you specify an action to take when the Simulator encounters each breakpoint. You can set more than one breakpoint on the same location, so that more than one action can be taken.

To set a breakpoint with conditions:

1. From the **Execute** menu, choose **Breakpoints** then select **Set**.
2. Under **Breakpoint Number** select the number you want to assign to this breakpoint. The default number that is shown is the next available number. Breakpoint numbers do not have to be consecutive; they can be assigned arbitrarily.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For example, it may be convenient to allocate breakpoints so that one function is assigned breakpoints 1 to 10, another uses 11 to 20, and so on.

3. Under **Type** select whether you are setting a breakpoint at a particular memory location, register, or expression.
4. If the breakpoint is being set for a memory location or a register, under **Access** specify what kind of access should be detected by the breakpoint. For example, if you want the breakpoint to detect when a memory location is read but not written to, select **Read**. If you want either a read or a write to be detected, chose **Read/Write**.
5. If the breakpoint is being set on a memory location, under **Memory** select the memory space.If the memory address you want to break on is a single location (as opposed to a memory block) type in the address under **Start Address** and leave the **End Address** blank.If you want to set the breakpoint on a range of addresses, type in the **Start Address** and **End Address** of the range. Setting a range means that the breakpoint will be recognized if any address in the range is accessed.You can also define a line number in the source program as a breakpoint. (The source program must contain symbolic debug line number information.) Specify the line number under **Start Address** in the form: *filename@line_number*. For example, to set a breakpoint on line 22 of the source file named clrmem.cld, under the **Start Address** you type: `clrmem@22`
6. If the breakpoint is being set on a register, under **Register** select the register.
7. If the breakpoint is being set on an expression, under **Expression** type the expression. Remember, if the expression is a C expression, enclose the expression in curly bracket: {*c_expression*}. A breakpoint expression can be any logical expression that is valid for the Motorola DSP Assembler. Table 1-1 provides a list of operators that can be used in the breakpoint expression.

Table 1-1. List of Operators

Operator	Function
<	less than
&&	logical “and”
<=	less than or equal to
||	logical "or"
==	equal to
!	logical "negate"

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 1-1. List of Operators (Continued)

Operator	Function
>=	greater than or equal to
&	bitwise "and"
>	greater than
|	bitwise "or"
!=	not equal to
~	bitwise one's complement
+	addition
^	bitwise "exclusive or"
-	subtraction
<<	shift left
/	division
>>	shift right

8. The breakpoint expression usually involves comparison of register or memory values. You can use any register name in the expression. There are also two special flag variables that you may reference in the breakpoint expression:

eof Is TRUE if an end-of-file condition occurs in an input file assigned to a peripheral or memory location.

jump Is TRUE if a "jump" change of flow occurs during code execution.

9. Under **Action** select what action is taken when the breakpoint is encountered. Table 1-2 provides a list of breakpoint actions.

Table 1-2. Breakpoint Actions

Action	Behavior
Halt	Stops program execution when the breakpoint is encountered.
Note	Displays the breakpoint expression in the Session window each time it is true. Program execution continues. The display in the Session window is not updated until program execution stops.
Show	Displays the enabled register/memory set. Program execution continues.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 1-2. Breakpoint Actions (Continued)

Action	Behavior
`Command`	Executes a Simulator command at breakpoint. Device execution commands, such as `trace` or `go`, will not execute.
`Increment[n]`	Increments the counter *n* variable by one.

10. If the action specified executes a command, under **Command** type the Simulator command. For example, a common command to perform in conjunction with a breakpoint is EVALUATE - evaluate an expression.
11. Click **OK**.

To clear a breakpoint:

1. From the **Execute** menu, choose **Breakpoints** then select **Clear**. A dialog box displays a list of all the current breakpoints.
2. Select the breakpoint you want removed so that it is highlighted. If you are clearing consecutive breakpoints you can click and drag to highlight more than one breakpoint. Or hold down the CTRL key while clicking on breakpoints to select more than one.
3. Click **OK** to delete the breakpoints you selected. Breakpoints will not be renumbered. For example, if you have set breakpoints #1, #2, and #3 and then clear breakpoint #2, the Simulator will number the remaining breakpoints #1 and #3. (The Simulator does not renumber the breakpoints; it retains the original number you assigned to the breakpoints.)

.

For more detailed information, see Section 9.1, "Introduction to Expressions," on page 9-1, and Section 11.1, "Display the Current Breakpoints," on page 11-1. Section 12.5, "break - Set, Modify, or Clear Breakpoint," on page 12-8 provides a complete command description.

1.10 Starting the Execution of Instructions with GO

The most common way to initiate program execution is with GO. When you indicate GO to the Simulator, it fetches, decodes, and executes instructions in the exact manner a device would if you were debugging an actual device. The GO command executes the program until one of the following occurs:

- the Simulator reaches a breakpoint

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

- you indicate STOP
- the Simulator encounters a debug instruction

To start executing instructions with GO:

1. From the **Execute** menu choose GO. You will see the following dialog box:

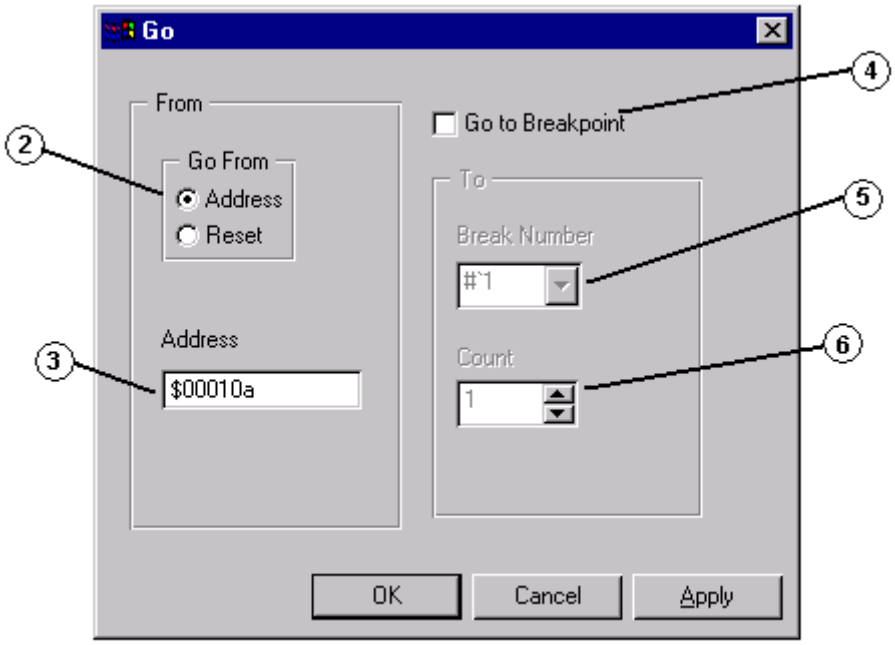

Figure 1-5. Execution Instructions with Go

3. Under **Go From**, choose whether to GO from an address or RESET. If you choose **Reset**, the Simulator simulates the reset sequence in the processor. The instruction pipeline, instruction counter, and cycle counter will be cleared before program simulation begins. The device registers are reset and execution begins at the reset exception address.
4. If you chose to GO from an address, under **Address**, type in the address from which you want to begin executing instructions. If you leave the address blank, execution will begin from the current program counter value. If you specify an address, the instruction pipeline, instruction counter, and cycle counter will be cleared before program simulation. Execution will begin from the address you specify.
5. Select the **Go to Breakpoints** checkbox if you want to stop execution at a particular breakpoint. If selected, all other stop breakpoints will be ignored.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

6. Under **Break Number** select the breakpoint where you want to stop execution. To see where you have breakpoints set, display the current breakpoints in the Breakpoint window.
7. Under **Count** specify how many times the Simulator should encounter the breakpoint before stopping. For example, if you set the count to 3, the program execution will be stopped on the third time that the breakpoint is encountered. The last instruction executed will be the breakpoint
8. When all conditions are set, Click **OK**. Execution begins.

Notice that during execution, the status bar in the lower left corner of the Simulator window shows the number of the device where execution is taking place, the PC (program counter), and the cycle count (updated every 1,000 cycles).

As an alternative, you can start program execution using the **GO** Button located on the toolbar.

Fore more details, see Section 2.9, "Resetting the Device," on page 2-7. Section 12.15, "go - Execute DSP Program," on page 12-22, provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 2
Controlling Execution

2.1 Starting the Execution of Instructions with GO

The most common way to initiate program execution is with GO. When you indicate GO to the Simulator, it fetches, decodes, and executes instructions in the exact manner a device would if you were debugging an actual device. The GO command executes the program until one of the following occurs:

- the Simulator reaches a breakpoint
- you indicate STOP
- the Simulator encounters a debug instruction

Section 1.10, "Starting the Execution of Instructions with GO," on page 1-13 provides a step-by-step task description for executing with GO from the GUI interface.

2.2 STEP Through Instructions

You can specify the number of instructions, lines, or cycles the Suite56 Simulator executes before stopping. Stepping through instructions is a quick way to specify the execution of a number of instructions without setting a breakpoint. The STEP instruction is similar to the TRACE instruction except that the display of the registers and memory blocks occurs only after the specified number of cycles, lines, or instructions have been executed.

If the next instruction to be executed calls a subroutine or begins execution of a macro, the subroutine or macro is not executed. However, it is possible to allow the current function to FINISH.

To step through program instructions:

1. From the **Execute** menu choose **Step**. You will see the following dialog box:

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

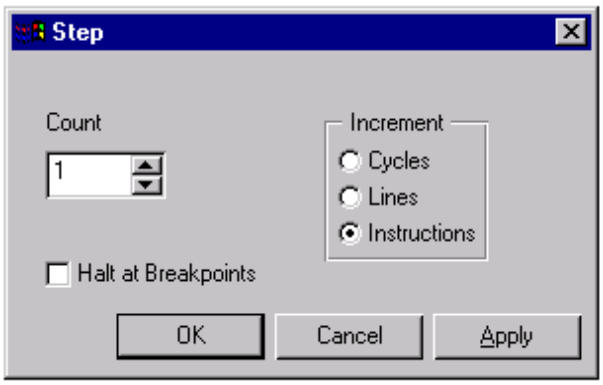

Figure 2-1. Step through Program Instructions

2. Under **Increment**, select whether to step by **Cycles**, **Lines**, or **Instructions**.
3. Under **Count** specify the number of cycles, lines, or instructions to step.
4. Select the checkbox labeled **Halt at Breakpoints** if you want breakpoints to be observed. If you do select the checkbox, the Simulator ignores breakpoints and does not halt program execution.
5. Click **OK**.

Notice that the Simulator updates the values in the Register window, the Memory window, and all other windows after it executes the last cycle, line, or instruction.

Alternatively, use the **STEP** Button on the toolbar to step one instruction or line at a time.

Section 12.31, "step - Step Through DSP Program," on page 12-41 provides a complete command description.

2.3 TRACE the Execution of Instructions

When you trace program execution, you can look at a snapshot of the enabled registers and memory after the Simulator executes each instruction or clock cycle.

To trace program execution:

1. From the **Execute** menu choose **Trace**. The following dialog box appears:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

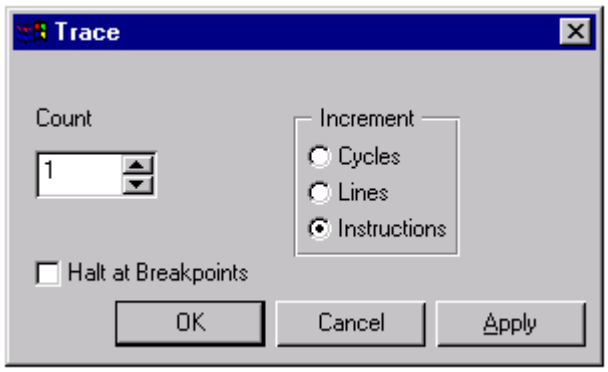

Figure 2-2. Trace Program Execution

2. Under **Increment** select **Cycle**, **Line**, or **Instruction**.
3. Under **Count** specify the total number of cycles, lines, or instructions to trace. Execution stops after this number of cycles/lines/instructions.
4. Select the checkbox labeled **Halt at Breakpoints** if you want to observe breakpoints.
5. If you do not select the checkbox, the Simulator ignores breakpoints and does not halt program execution.
6. Click **OK**.

Notice that the Simulator updates values in the register window, the memory window, and all other windows after it executes each cycle, line, or instruction. It updates these values at the speed of execution, so they will pass by very quickly. In order to review the values, scroll back through the Session window once execution has stopped.

It is often useful to log output from the Session window Log.

Section 12.34, "trace - Trace Through DSP Program," on page 12-44, provides a complete command description.

2.4 Executing the NEXT Instruction

You can specify the number of instructions or lines the Simulator executes before stopping. When you execute the next specific number of instructions you can quickly control execution without setting a breakpoint.

Executing instructions with NEXT is similar to performing a STEP through instructions, except that you can only specify the number of lines or instructions, and not cycles. Like stepping through instructions, the Simulator displays the registers and memory blocks only after it executes the specified number of lines or instructions.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

The important difference between NEXT and STEP is the way each instruction handles subroutines. If the next instruction executed calls a subroutine or begins execution of a macro, all the instructions of the subroutine or macro are executed before execution stops. The Simulator then displays enabled registers, memory, and other updated values. In order to recognize macros, the symbolic debug information for the program code must be loaded. The debug information is included in the COFF format .cld files generated using the Motorola Suite 56 DSP Assembler's `-g` option. Motorola's Suite56 C Compilers and the Star*Core 100 C/C++ Compiler also provide debug information when you use the -g option.

To execute the next program instruction:

1. From the **Execute** menu choose **Next**.
2. Under **Increment**, select to execute by lines or by instructions.
3. Under **Count**, specify the number of lines or instructions to execute.
4. Select the checkbox labeled **Halt at Breakpoints** if you want to observe breakpoints. If you do not select the checkbox, the instruction will ignore breakpoints and will not halt program execution.
5. Click **OK**.

Notice that the Simulator updates the values in the Register window, the Memory window, and all other windows after it executes the last line or instruction. Alternatively, you can use the **Next** button on the toolbar to execute the next instruction or line

Section 10.5, "Next Button," on page 10-4 provides more advanced information.
Section 12.23, "next- Step Over Subroutine Calls or Macros," on page 12-32 provides a complete command description.

2.5 Providing an UNTIL Condition

An UNTIL condition causes the Simulator to execute the currently loaded program until it encounters a specified location. You can specify the location as a program line number, an address, or a label. An UNTIL condition works essentially as a temporary breakpoint, cleared when execution stops.

To provide an UNTIL condition:

1. From the **Execute** menu, choose **Until**.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

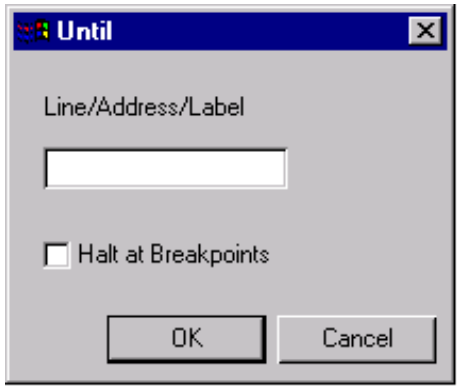

Figure 2-3. Providing an Until Condition

2. Type the line address or label at which to stop execution. (You can only use line numbers and labels if debug information has been loaded from a COFF file.) To specify an address, include the memory space followed by a colon, followed by the address. For example, to execute instructions until address `$00c103` is reached, type:

   ```
   p:$00c103
   ```

   To specify a line number in the currently loaded source file, just type in the line number. Or to specify a line number of a particular source file, include the filename. For example, to execute until line number 20 of a source file named clrmem.cld located in the working directory, type:`clrmem@20`
3. Select the checkbox labeled **Halt at Breakpoints** if you want the instruction to observe breakpoints. If you do not select the checkbox, the instruction ignores breakpoints and does not halt program execution.
4. Click **OK**.

Section 12.37, "until - Execute Until Address," on page 12-46 provides a complete command description.

2.6 Pausing Execution with WAIT

You can pause macro execution by specifying the number of seconds you want the Simulator to wait. Execution resumes after the pause.

The WAIT instruction is particularly useful when you write a command macro. That is, if you cause the simulation to WAIT while logging commands, you can then save the log file. A command macro results. The pause that was recorded while logging commands will be replayed when the macro is executed. The wait instruction provides a useful way to pause execution so that you can examine certain values or windows.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

To pause execution:

1. From the **Execute** menu, choose **Wait**.
2. Select the number of seconds to pause or click on the checkbox **Forever** to pause indefinitely.
3. Click **OK**.
4. An information box appears to let you know that the Simulator is paused. If you don't want to wait for the pause to end automatically, you can resume program execution by clicking **Cancel**. Keep in mind that if you click cancel while recording a command macro that the cancellation of the pause is not recorded. So, for example, if you specify a WAIT of ten seconds and then cancel after three, when the command macro executes, it will pause for the full ten seconds.

Section 8.1, "Creating and Running a Command Macro," on page 8-1 provides more detailed information. Section 12.40, "wait - Wait Specified Time," on page 12-48 provides a complete command description.

2.7 Allowing the Current Function to FINISH after an UNTIL Condition or a Breakpoint

Sometimes a program execution stops while in the middle of executing a subroutine or function. This might occur for several reasons. Execution may stop if you specify that a certain number of steps be executed which happened to end in the middle of the subroutine. Or, execution may stop if an UNTIL condition or breakpoint is reached while in the middle of a subroutine. Section 1.9, "Setting and Modifying Breakpoints," provides a detailed discussion of breakpoint funtions.

In any instance where the Simulator stops execution in the middle of a subroutine or function, the subroutine or function can be made to finish execution.

To finish execution of the current function or subroutine:

1. From the **Execute** menu, choose **Finish**. The Simulator continues until it encounters an RTS instruction. Execution continues only to the end of the current function. If the Simulator encounters another function during a FINISH operation, the execution of the encountered function does not complete.

Alternatively, you can use the **Finish** button on the toolbar.

Section 12.13, "finish - Execute Until End of Current Subroutine," on page 12-21 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

2.8 Stopping Program Execution

You can stop program execution or the execution of a command macro at any time.

To stop program or command macro execution:

1. From the **Execute** menu, choose **Stop**: notice that this is the only menu item you can choose while program or command macro execution is in progress. If you began the execution by providing an UNTIL condition, the Simulator will clear this temporary breakpoint. The Assembly window (and the Source window if applicable) highlight the next instruction to be executed in red.

When program execution is stopped the Session window will display **Simulation Aborted**. Notice that the Simulator updates the values in the Session window, the Register window, the Memory window, and all other windows to reflect the last instruction or line executed.

Alternatively, you can use the **Stop** button on the toolbar to stop program execution.

For more details about stopping program execution, see Section 2.9, "Resetting the Device," and Section 10.8, "Repeat Button," on page 10-7.

2.9 Resetting the Device

You can reset the device or the entire Simulator state any time instruction execution is halted. Resetting the device also allows you to select the operating mode that the device will be set to in response to a simulated hardware reset sequence.

To reset the device:

1. Before you reset the device you will want to know which operating modes are available to the device that you are simulating. To see the available operating modes, open the Command window. At the command line, type:
   ```
   help mode
   ```
2. View the Session window. It now contains a list of the operating modes available for the device that you are simulating. Now that you know the operating modes from which you can choose, proceed in resetting the registers.
3. From the **Execute** menu choose **Reset**, then select **Device**. The following dialog box appears:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

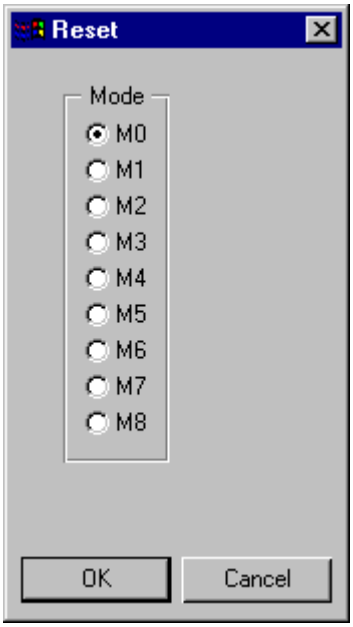

Figure 2-4. Resetting the Device Registers

4. Select the operating mode to which the device should be reset.
5. Click **OK**.

To reset the simulation memory:

1. From the **Execute** menu choose **Reset**, then select **State**. This resets the entire Simulator state to the start-up condition. All breakpoints are cleared, the memory is initialized, and all logging and I/O files are closed.

Section 12.29, "reset - Reset Device or State," on page 12-39 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 3
Object Files and Data Files

To efficiently manage and use the object files and data files in your project, you will need to understand how the Simulator sets and clears paths. For a complete discussion of the working directory path and alternate directory paths, see Section 1.5, "Setting and Clearing the Path."

3.1 Displaying the Current PATH

The DSP Simulator uses two types of paths to save and access files:

- the working directory
- alternate directories

The Simulator uses the working directory as the default directory when searching for an input file (assuming no path is explicitly specified with the input filename). Alternate directories are also searched, in turn, if an input file is not found in the working directory.

To display the current paths:

1. From the **Display** menu choose **Path**
2. Open the Session window if not already open. The Simulator displays the current working directory and the alternate source paths.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

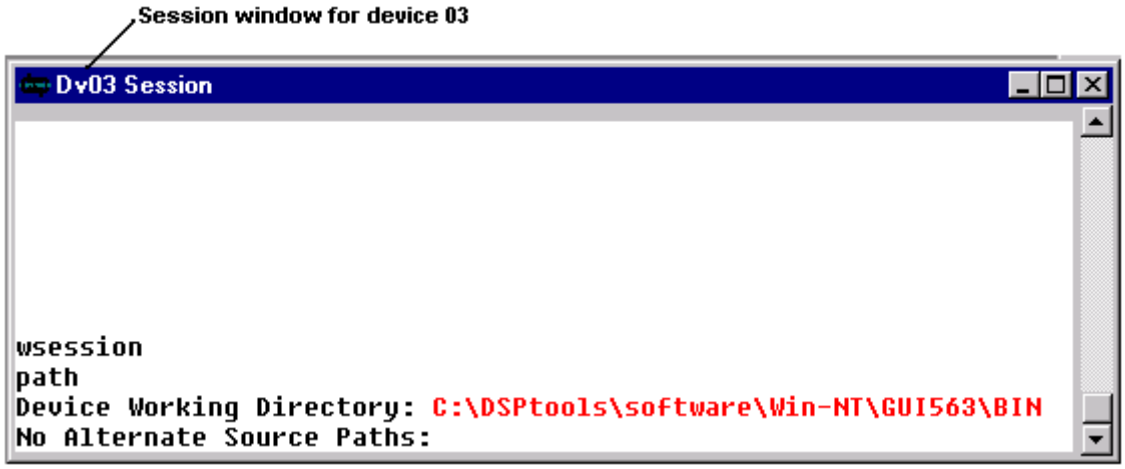

Figure 3-1. Displaying Current Paths

Keep in mind that the Simulator displays a separate directory path for each device. This means that as you switch from one device to another, the working directory also changes according to the current device. The top left corner of the Session window indicates the current path.

Section 12.25, "path - Specify Default Pathname," on page 12-35 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

3.2 Saving Object Files

You can save memory blocks in DSP COFF or ASCII OMF object files. The object files can be reloaded in the Simulator or in any other environment where such files are used. For a complete description of the steps required to load object files in either COFF or OMF format, see Section 1.6, "Loading Object Files," on page 1-6.

To save a COFF (.cld) file:

1. From the **File** menu choose **Save** then select **Memory COFF**.
2. Under **Memory Space**, select the memory to save.
3. Under **Start Address**, type the beginning address of the memory block.
4. Under **End Address**, type the last address of the memory block.
5. Under **File Name**, type in the filename of the file to save or click on **File** to browse the directories. The extension default is .cld.
6. Click **OK**. If no directory path is specified, the file will be saved in the working directory. If the file already exists, you will be prompted to decide whether to overwrite the file or to append it.

The Simulator does not store symbolic debug information.

To save an OMF (.lod) file:

1. From the **File** menu, choose **Save** Then select **Memory OMF**.
2. Under **Memory Space**, select the memory to save.
3. Under **Start Address**, type the beginning address of the memory block.
4. Under **End Address**, type the last address of the memory block.
5. Under **File Name**, type in the filename of the file to save or click on **File** to browse the directories.The extension default is .lod.
6. Click **OK**. If no directory path is specified, the file will be saved in the working directory. If the file already exists, you will be prompted to decide whether to overwrite the file or to append it.

Section 12.30, "save - Save Simulator File," on page 12-40 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 4
Managing Memory and Registers

To efficiently manage the memory and registers associated with your project, you will need to know how to reset the device. For a complete discussion of the resetting the device, see Section 2.9, "Resetting the Device."

4.1 Displaying Register Values

You can display the device registers and memory any time instruction execution is halted. To display the value in a register:

1. From the **Windows** menu choose **Register**. The following dialog box appears:

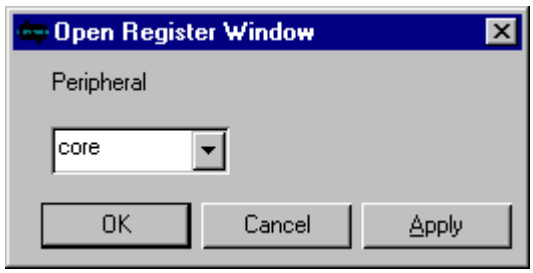

Figure 4-1. DSP Simulator Register Dialog Box

2. Select the peripheral that you want to view.
3. Click **OK**.

The register values for the peripheral that you chose appear in a register window: For example, if you choose to display the register values for the core, you see a window similar to the following:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

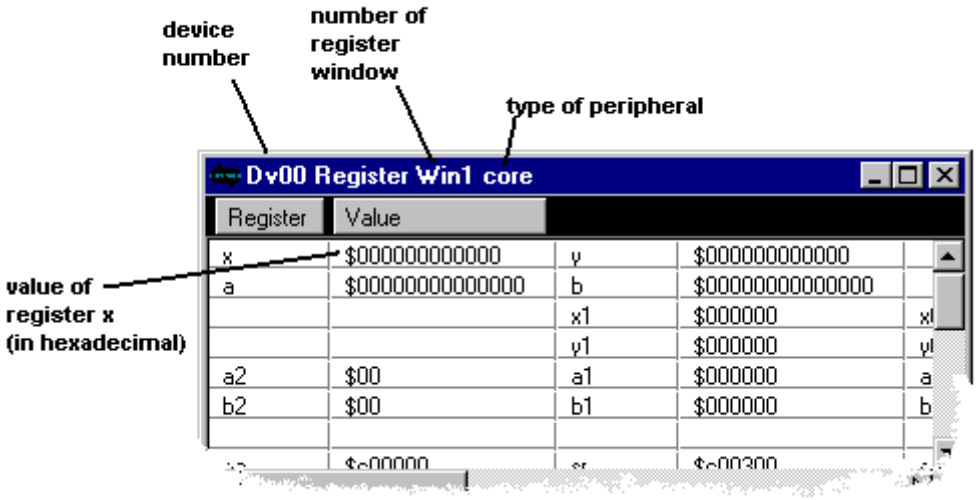

Figure 4-2. Displaying the Register Values

Notice that the title bar of the register window indicates:

- the device number where the peripheral resides,
- the number of the Register window. This number is shown because you can display other Register windows for other peripherals.
- the type of peripheral. The peripheral type is shown for easy identification. Several register windows can be open – each corresponding to a different peripheral.

Section 12.10, "display - Display Register or Memory," on page 12-17 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

4.2 Changing Register Values

You can change the value of the device registers when instruction execution is halted.

To change the value of a specific register:

1. From the **Modify** menu choose **Change Register**. The following dialog box appears:

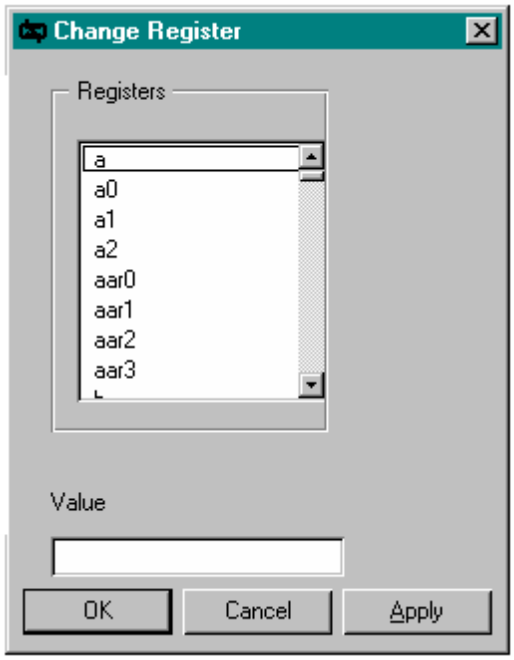

Figure 4-3. Changing the Value of Register

2. Select the register whose value you want to change.
3. Under **Value**, type in the value to which you want the register to be set
4. Click **OK**.

Section 12.6, "change - Change Register or Memory Value," on page 12-13 provides a complete command description.

4.3 Displaying Memory Values

You can display the values in memory any time instruction execution is halted.

To display the value in memory:

1. From the **Windows** menu choose **Memory**. The following dialog box appears:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

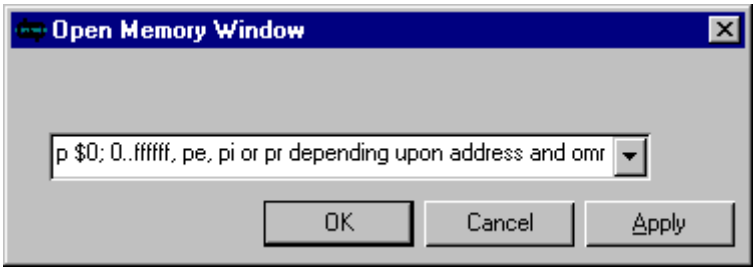

Figure 4-4. DSP Simulator Memory Dialog Box

2. From the drop-down list box, select the memory space whose values you want to change.
3. Click **OK**.

The memory values for the memory space you choose appear in a memory window. For example, if you display values in the p memory space, you see a window similar to the following:

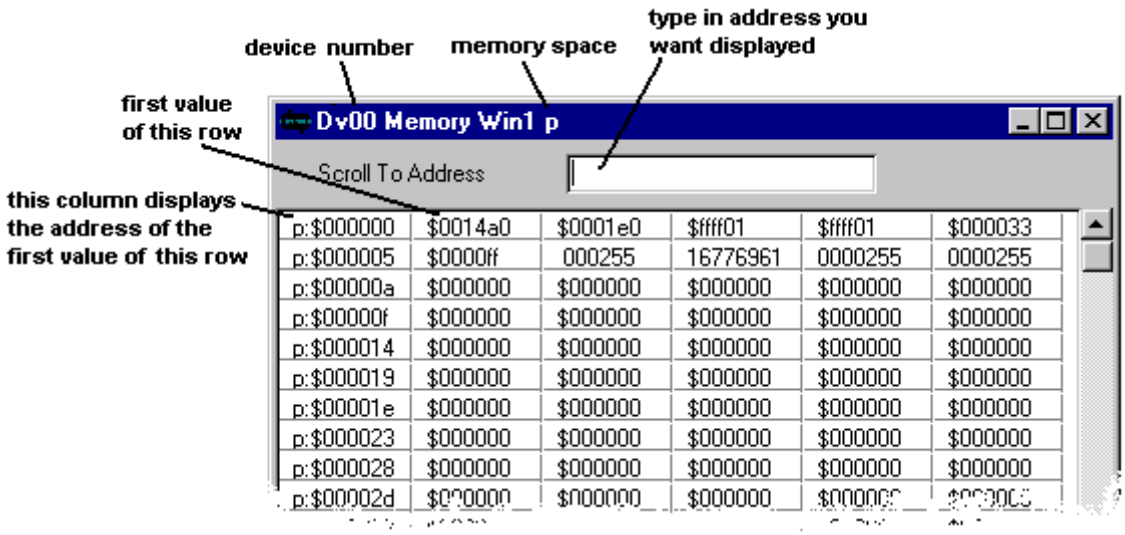

Figure 4-5. Displaying Memory Values

Notice that the title bar of the memory window indicates:

- the device number that you are viewing,
- the number of the memory window. This is shown because you can display other memory windows simultaneously.
- the type of memory space. Again, you could have several memory windows open – each corresponding to a different memory space.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Section 12.10, "display - Display Register or Memory," on page 12-17 provides a complete command description.

4.4 Changing Memory Values

You can change the values in memory any time instruction execution is halted.

To change the value in memory:

1. From the **Modify** menu choose **Change Memory**. The following dialog box appears:

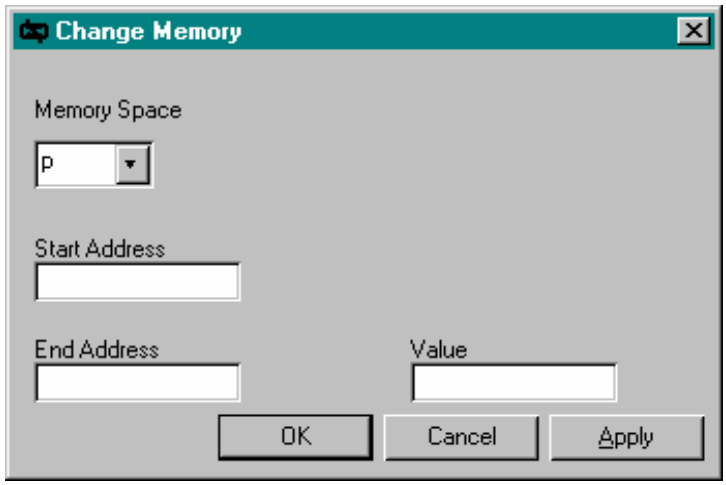

Figure 4-6. Changing the Value in Memory

2. Select the memory space that you want to change.
3. Under **Start Address**, type the value of the beginning address that you want to change.
4. Under **End Address**, type the value of the ending address that you want to change.
5. Under **Value**, type the value to which the specified addresses should be changed. Keep in mind that all values, including the starting and ending values will be changed to the value that you specify.The format of this value (HEX, decimal, etc.) will automatically be the same as the current default radix. If you want the format of the value to differ from the default, you must use the appropriate radix specifier.

   $ denotes value as hexadecimal

   ‘ denotes value as decimal

   % denotes value as binary

6. Click **OK**.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Section 12.6, "change - Change Register or Memory Value," on page 12-13 provides a complete command description.

4.5 Copying Memory from One Block to Another

You can copy memory blocks from one location to another. The source and destination memory maps may be different. This allows you to move data or program code from one memory map to another or to a different address within the same memory map.

To copy memory from one block to another:

1. From the **Modify** menu choose **Copy Memory** The following dialog box appears:

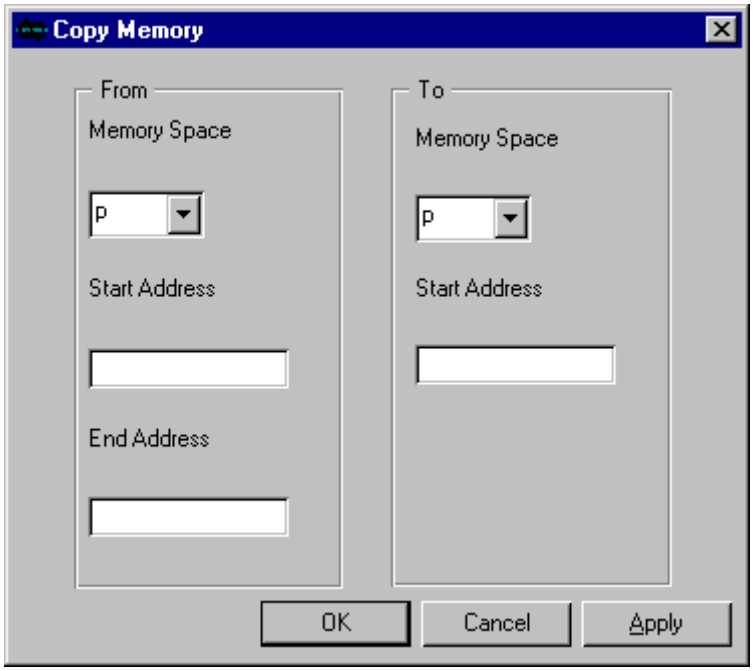

Figure 4-7. Copying Memory from One Block to Another

2. In the **From Memory Space** pane, select the memory space to copy from. Then in the text boxes, type the starting address and the ending address of the block that you want copied.
3. In the **To Memory Space** pane, select the memory space to copy to. Then type the starting address to begin copying values.
4. Click **OK**.

Section 12.7, "copy - Copy a Memory Block," on page 12-14 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

4.6 Disassembling Code Stored in Memory

You can disassemble instructions that have been loaded so that you can review DSP object code in its assembly language mnemonic format. Invalid opcodes are disassembled to a define constant (DC) mnemonic.

To disassemble code stored in memory:

1. From the **Display** menu choose **Disassemble** then select **Memory Block**. The following dialog box appears:

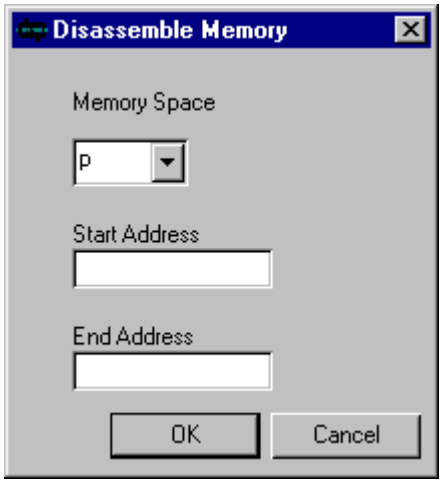

Figure 4-8. Disassembling Code Stored in Memory

2. Select the memory space that you want to disassemble.
3. In the **Start Address** text box, type the address where you want to begin to disassemble.
4. In the **End Address** text box, type the address where you want to stop disassembling.
5. Click **OK**.
6. View the Session window to see the mnemonics of the memory block you specified.

Section 12.9, "disassemble - Object Code Disassembler," on page 12-16 and Section 12.27, "radix - Change Input or Display Radix," provide complete command descriptions.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

4.7 Displaying a History of Recent Instructions

You can disassemble and display a history of the most recent instructions executed by the device.

A typical use for displaying the history would be to determine the sequence of instructions that terminated in a user-defined breakpoint. You would set the breakpoint conditions, then issue the GO command. When the break condition is met, instruction execution halts and the currently enabled registers appear. You can then display the history to view the last 32 instructions that executed prior to the breakpoint.

To display a history:

1. From the **Display** menu choose **History**.
2. View the Session window to see the last 32 instructions executed by the device.

The instructions appear in the order executed, with the most recent instruction appearing at the bottom of the list. The last instruction in the list has been fetched and decoded by the device and enters the execute phase in the next device cycle. It is in the same state as instructions that are disassembled and displayed at the end of each trace display.

The device execution history can also be logged continuously to an output file using the Simulator output command.

The device execution history can also be logged continuously to a file by assigning an output file. When you assign the output file, choose to output from extended history. Remember to close the file before viewing its contents. Section 5.12, "Assigning an Output File," on page 5-11provides more detailed information.

Section 12.17, "history -Disassemble Previously Executed Instruction," on page 12-24 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 5
Device I/O and Peripheral Simulation

5.1 Introduction to Device I/O (Input/Output)

The Suite 56 DSP Simulator provides you with the ability to simulate the data input and output. If you were actually debugging a DSP device, you would need several pieces of hardware. The hardware includes an actual device and peripheral devices that supply input to the DSP device and recognize output from the DSP device.

In the Simulator environment, of course, you are not debugging an actual device, which means that there is no communication with peripherals, ports, or pins. Instead, the Simulator provides simulated input and output by using I/O files. These I/O files communicate with the Simulator as if the input and output were coming from an actual peripheral, port, pin, or memory location.

You can specify whether the input data is to come from a file or from the keyboard (terminal). The Simulator simulates DSP on-chip peripherals on a cycle by cycle basis.

5.2 How Are I/O Files Formatted?

Simulated input and output is represented in ASCII format, which means that you can conveniently edit and print I/O files from a text editor. An input file or output file can contain:

- repeat punctuation
- comments
- timing information
- peripheral data
- port data
- pin data
- memory data

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

5.3 Repeat Punctuation

The Simulator provides a way to specify repeated input or output data values and sequences. A single data value can be repeated by adding the following syntax after a data item:

`#{`*`count`*`}`

Enclosing several data items in parentheses indicates that the data items are to be treated as a group. The entire group can then be repeated by placing `#{`*`count`*`}` immediately following the closing parenthesis. The parentheses can be nested. A closing parenthesis without a following repeat count causes the data sequence within the parentheses to repeat forever. Timed values can appear within a repeat group, and in this case the relative time mode (`+time`) should be used.

Table 5-1. Repeat Punctuation Input Data

Data Sample	Explanation
`1FF#20`	Repeats the untimed data item 1FF twenty times.
`(+5 CC +10 33)#5`	Repeats the sequence of timed data pairs +5 CC +10 33 five times.
`(CC354 CC333 C7000)`	Repeats the untimed data sequence CC354 CC333 C7000 forever.
`(1#5 0#5)`	Repeats the untimed data sequence 1 1 1 1 1 0 0 0 0 0 forever.

5.4 Comments

Any information following a semicolon, up to the end-of-line, is considered to be a user comment and is not interpreted as input data or timing.

Example 5-1. Comment Code

```
FFC 333 972 ;next three p memory data words
```

In this example, the first three data values are applied to the device. The information following the semicolon is a user comment.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

5.5 Timing Information

If the `t` key character is specified in the input or output command, then the assigned file will contain cycle timing information preceding each piece of I/O data. The timing information relates to the Simulator cycle counter value (cyc register) at the time when the data transfer occurs. The timing information is always expressed in decimal. If the timing information is preceded by a plus sign (+), it indicates a relative number of cycles from the preceding specified timing value; otherwise it indicates the exact value of the Simulator cycle register at the time of the transfer.

5.6 Peripheral Data

You can assign and input or output file to each DSP peripheral. The following general information applies to all peripheral files.

The peripheral data value can be represented in hexadecimal, decimal, binary or floating point. Specify the default input radix in the input command; specify the output radix in the output command.

You can express floating point input in the usual methods.For example,

```
0.5
5e-1
5.0E-1
```

are all acceptable data input values. If a data value contains a decimal point, the data will be input as a floating point value, overriding the input radix specification. The number base of data values can also be denoted by preceding the data value with a dollar sign ($), an apostrophe (‘), or a percent sign (%).

`$`	input as hexadecimal
‘	input as decimal
`%`	input as binary

Untimed peripheral input data values are applied only during cycles when the peripheral function is enabled. Some peripherals retrieve data from the input file when the peripheral would normally receive new data; other peripherals retrieve the data each cycle. The final specified data value remains applied to the peripheral indefinitely. The repeat punctuation and repeat count can specify durations of longer than one cycle.

Timed peripheral input data values will be applied to the peripheral at the time intervals, or at the exact Simulator cycles indicated by the timing information within the file. If the

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

first timing information in the file is a relative value, (timing preceded by +) the Simulator will wait until the peripheral function is enabled before applying the data.

If a lower case letter `t` is placed in a data position of the input file, you will be prompted for the next input data value as described below in Section 5.10, "Terminal Input of Data Values."

Storage of data to the peripheral output file will begin when the peripheral is enabled. In the timed output mode, the Simulator cycle count and a data value are stored each time the peripheral output changes. In the untimed output mode, a data value and a following repeat count are stored each time the data changes.

5.7 Port Data

When assigned to a DSP port, the input file data value represents the value applied to all the pins of the port. The least significant port bit maps to the least significant bit of the data value. The following general information applies to all port files.

A port is simply a convenient grouping of device pins. Untimed data applied to a port is retrieved each clock cycle, with one exception: the data bus ports retrieve new data from an assigned input file only once for each memory fetch.

The port data value can be represented in hexadecimal, decimal, binary or floating point. The default input radix can be specified when assigning an input file; the output radix can be specified when assigning an output file.

Floating point input can be expressed in the usual methods. For example,

```
0.5
5e-1
5.0E-1
```

are all acceptable data input values. If a data value contains a decimal point, the data will be input as a floating point value, overriding the input radix specification. The number base of data values can also be denoted by preceding the data value with a dollar sign (`$`), an apostrophe (‘), or a percent sign (`%`).

`$`	input as hexadecimal
‘	input as decimal
`%`	input as binary

Timed port input data values are applied to the port at the specified relative time intervals (`+time`), or at the exact Simulator cycle indicated by the timing information within the file.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

If a lower case letter `t` is placed in a data position of the input file, you will be prompted for the next input data value as described below in Section 5.10, "Terminal Input of Data Values."

Storage of data to a port output file will occur any time a write operation occurs to the port. In the timed output mode, the Simulator cycle count and a data value are stored each time a word is written. In the untimed output mode, a single data value is stored each time a word is written. No tristate information is stored in the port output data.

5.8 Memory Data

When assigned to a memory location, the input file data value supplies the value that is read when the Simulator references that memory location. The least significant memory bit maps to the least significant bit of the data value.

The input data value can be in decimal, binary, hexadecimal, or floating point form. The Simulator interprets the data based on the input radix specified in the input command. The default input radix is hexadecimal. If a data value contains a decimal point, the data will be input as a floating point value, overriding the input radix specification. The number base of data values can also be denoted by preceding the data value with a dollar sign ($), an apostrophe (‘), or a percent sign (%).

$	input as hexadecimal
‘	input as decimal
%	input as binary

The Simulator applies untimed memory input data values each time the device performs a read operation on the memory location. In other words, the input file acts like a stack of input data; each successive data value is retrieved from the “stack” file when a read operation occurs.

The Simulator applies timed input data values to the memory location at the specified relative time intervals (`+time`), or at the exact Simulator cycle indicated by the timing information within the file. If the first timing information in the file is a relative value (`+time`) the Simulator waits until the first read of that memory location before getting the first data value. Otherwise the data application occurs at the exact specified cycle.

If a lower case letter `t` is placed in a data position of the input file, you will be prompted for the next input data value as described below in Section 5.10, "Terminal Input of Data Values."

Storage of data to a memory output file occurs any time a write operation occurs to the memory location. In the timed output mode, the Simulator cycle count and a data value are

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

stored each time a word is written. In the untimed output mode, a single data value is stored each time a word is written.

The output data value can be in decimal, binary, hexadecimal, floating point or string form. Specify the output radix in the output command. The default output radix is hexadecimal. The string form of output data uses the value written to the memory location as the starting address in the same memory space of a zero terminated ASCII character string. The character string is written to the output file.

Table 5-2. Input Memory Data

Data Sample	Explanation
`7FFF 7F3F 5D3C 7FC3`	The untimed memory input file causes the data sequence 7FFF 7F3F 5D3C 7FC3 to appear during consecutive reads of the specified memory location.
`(1FF 0)`	The untimed memory input file causes the data sequence 1FF 0 to appear repeatedly during consecutive reads of the specified memory location.
`(+10 0.5 +10 0.3)`	The timed memory input file causes the data sequence 0.5 0.3 to alternate in the specified memory location 10 cycle intervals.
`2000 1C3 2005 1CF`	The timed memory input file causes 1C3 to appear in the specified memory location at cycle 2000, and 1CF to appear at cycle 2005.

5.9 Pin or Pin Group Data

When assigned to a pin or pin group, the input file data value supplies the zero (`0` or `L`), one (`1` or `H`), a negative pulse within a single cycle (`N`), a positive pulse within a single cycle (`P`) , or a tristate (`X`) value to be applied to the pin or to each pin in the group, with the first specified pin mapping to the least significant bit of the data value. Each data word must contain as many characters (`0`, `1`, `L`, `H`, `N`, `P`, or `X`) as there are pins in the group.

If an analog input file is assigned to a device analog pin, the input command must specify the floating point radix with the `-rf` radix designator in the input command. Likewise, an analog output pin file must be specified with the `-rf` radix designator in order to generate floating point output for the pin rather than the digital pin values described in the preceding paragraph. Floating point io files may only be assigned to a single analog pin.

The Simulator applies untimed pin input data values to the specified pin in each Simulator clock cycle. Timed pin input data values will be applied to the specified pin at the specified relative time intervals (`+time`), or at the exact Simulator cycle indicated by the timing information within the file.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

If a lower case letter `t` is placed in a data position of the input file, you will be prompted for the next input data value as described in the section below titled, Terminal Input of Data Values.

Storage of pin data to an output file will occur any time the pins data value changes, including changes to tristate, 1, 0, H or L. In the timed output mode, the Simulator cycle count and a data value are stored each time a word is written. In the untimed output mode, a single data value is stored each time a word is written.

The Simulator also provides a special mode that allows pin data input reception from the output of another pin without the necessity of an intermediate disk file.

Table 5-3. Pin or Pin Group Input Data

Data Sample	Explanations
`0 0 1`	This untimed Reset pin input file causes the Reset pin to go low for two cycles, then back high.
`12000 0 12010 1`	This timed IRQA pin input causes the IRQA pin to go low at cycle 12000, then back high at cycle 12010.
`(+200 0 +20 1)#9`	This timed IRQB pin input causes the IRQB pin to go low after 200 cycles and stay low for 20 cycles. The sequence is repeated 9 times.

5.10 Terminal Input of Data Values

The Simulator provides two levels of terminal data input. If the input command specifies `term` as the input filename, the Simulator opens an editor, which allows creation of an input data file without leaving the Simulator. The data file is given a temporary name, termxxxx.io or termxxxx.tio (xxxx being a numeral between 0000-9999), and is saved on the disk at the termination of the input command. You can specify the entire contents of the input file in this manner.

A second level of terminal data input allows you to be prompted any time the next input data value is needed. This method is triggered if the lower case letter `t` is encountered in the data field of the input file. This is only valid for the data field, not for the time field. Each time a `t` is encountered, you will be prompted for a single data value from the terminal. The Simulator will read the input data using the radix specified in the input command. Hexadecimal is the default input radix. If you just type the return key at the prompt, without entering a data value, the previous data value will be repeated. If you type the ESC key at the prompt, an end-of-file status will be simulated and the previous data value will repeat forever.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 5-4. Terminal Input Data

Data Sample	Explanations
`(t#45)`	This untimed IRQA pin input file prompts you for a new input value every 45 clock cycles.
`ffcc c1000 t ab12 t 6444`	This untimed memory file input data prompts you for the third and fifth values that are read from the specified memory location.
`(+10 5555 +10 3333)#3 566 t 800 t`	This timed port input file prompts you for port input data at cycle 566 and 800 after alternating the input data sequence 5555 3333 three times at ten cycle intervals.

See Section 6.3, "Changing the Radix," on page 6-2, for more advanced information about specifying your radix.

5.11 Assigning an Input File

By providing an input file, you can retrieve simulated peripheral, memory location, or device pin data.

If you are not sure about the valid peripheral and port names for the device that you are working on, use the Simulator's "`help periph`" and "`help port`" commands in the Command window to obtain a list of the peripherals and ports.

To provide input data:

1. From the **File** menu choose **Input** then select **Open**. The following dialog box appears:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

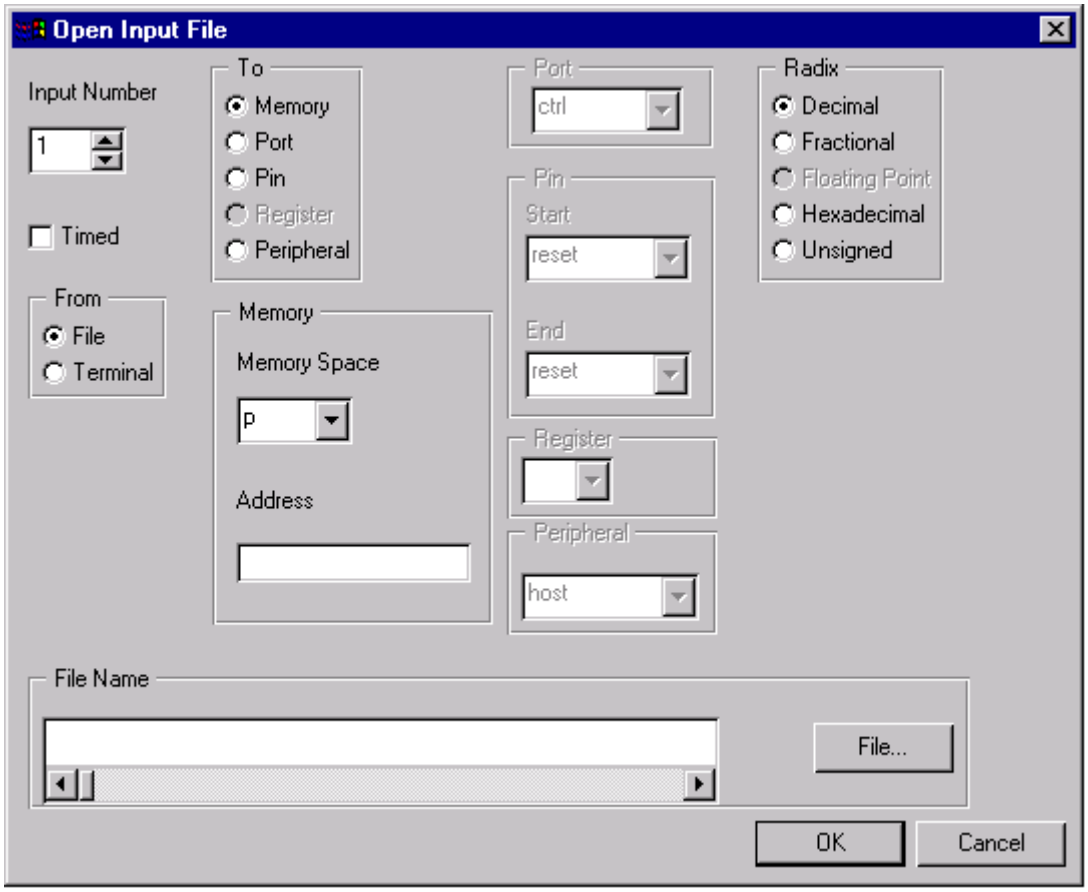

Figure 5-1. Providing Input Data

2. Under **Input Number** select the number you want to assign to this input. The default number shown is the next available number.Input numbers do not have to be consecutive— they can be assigned arbitrarily, which means that you can assign input numbers in any way that helps you organize the sources of the inputs.
3. Mark the checkbox labeled **Timed** if the data contains time-data pairs.
4. Under **From** select the source of the input data. Select **File** if the input data comes from an existing file. Select **Terminal** if you will enter the input data directly from the keyboard.
5. Under **To** select the data destination: **Memory Space**, **Port**, **Pin**, **Register**, or a **Peripheral**
6. If the input data is being sent to a memory space, under **Memory** select the memory space. Then type the desired address in the **Address** text box. Assigning data to a memory address causes all subsequent reads of that memory address to reference the input source. You can use this method to simulate your unique

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

memory mapped peripherals, or to short-circuit the simulation of the on-chip peripherals.

7. If the input data is being sent to a port, under **Port** select the name of the port. The list of available ports will vary depending on the device that you are targeting. To see a list of port names and their index and mask, type `help port` from the command line of the Command window .
8. If the input data is being sent to a pin, under **Pin** select the pin location. If the input receiving pin is a single location (as opposed to a range of pins) select the name of the pin under **Start**. If you want the input data to go to a range of pins, select the names of the **Start** and **End** pins in the appropriate drop-down box. To see a list of pin names and their indices, type `help pin` from the command line of the Command Window.
9. If the input data is being sent to a register, under **Register** select the register.
10. If the input data is being sent to a peripheral, under **Peripheral** select the name of the peripheral. The list of available peripherals varies depending on the device that you are targeting. To see a list of peripheral names with indices type `help periph`, from the command line of the Command window .
11. Under **Radix** select the input data's number system (hexadecimal, decimal, etc.).
12. If the input data is coming from a file, under **File Name** type in the name of the input file or click on the **File** button to browse for the file. The default filename extension is `.io`.The data file may contain only data, in which case each access to the object returns the next value in the data file. Alternatively, the file may contain time/data pairs. In this case, each pair specifies the value to input at or after the specified cycle count. Repeated accesses will return the same value until the simulated cycle count reaches the time specified in the next time/data pair.
13. Click **OK**.If you are entering the data from the terminal (keyboard), the **Interactive Input** dialog box appears, which prompts you for the first data value. Type in the first data value and click **OK**. After you enter a value and click **OK**, the **Interactive Input** dialog box appears, ready for the second data value. Continue to enter values and click **OK** until you have entered all values. When you have entered all values, click **Cancel** in the **Interactive Input** dialog box. The Simulator creates a file in the working directory that contains the data you entered from the keyboard. The filename will be similar to term0000.io. You can reuse this file as you would any other input file.

Section 12.18, "input - Assign Input File," on page 12-24 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

5.12 Assigning an Output File

You can write a single data item from the default device to a file or to the session window (terminal). The data item may be a single memory location, a port, a range of pins, a peripheral, or execution history. You must establish a separate output file for each data item written.

To create an output file:

1. From the **File** menu choose **Output** then select **Open**. The following dialog box appears:

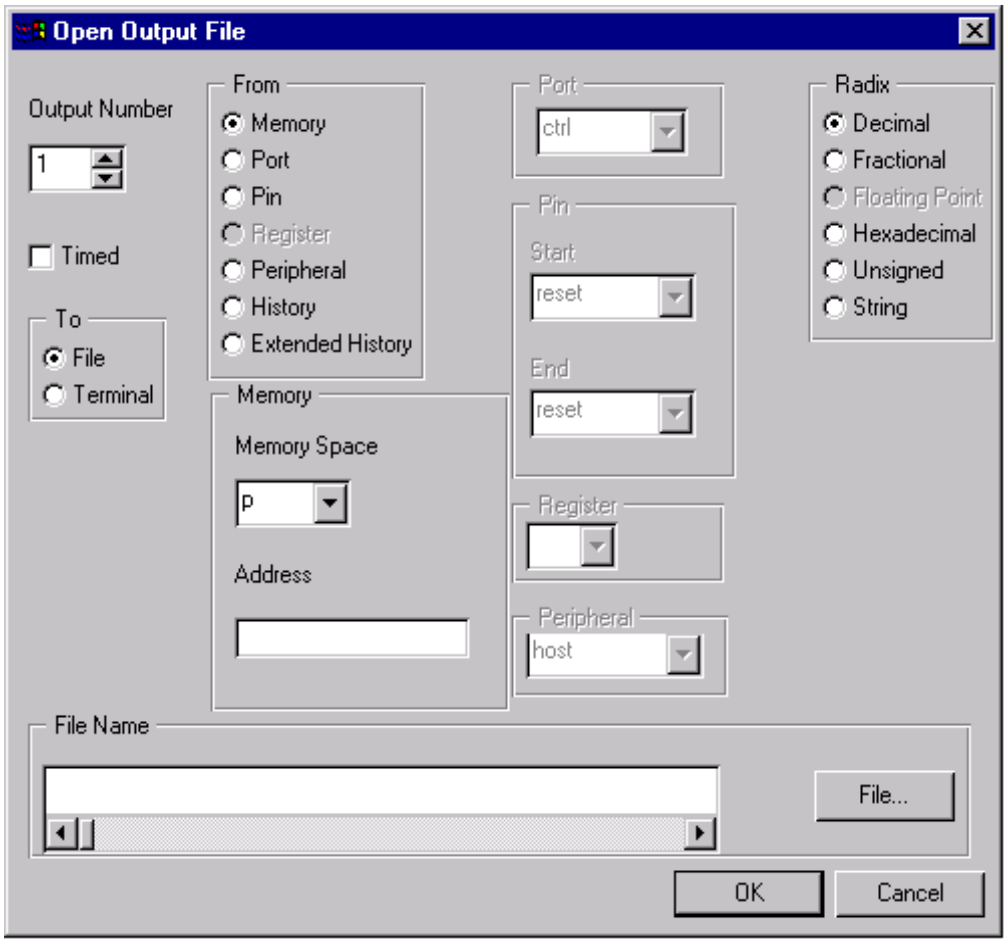

Figure 5-2. Creating an Output File

2. Under **Output Number** select the number you want to assign to this output. The default number shown is the next available number. Output numbers do not have to be consecutive; they can be assigned arbitrarily, which means that you can assign output numbers in any way that helps you organize the output files.
3. Mark the checkbox labeled **Timed** if the data contains time-data pairs

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

4. Under **To** select the destination of the output file Select **File** if the output data is to be written to a file. Select **Terminal** if you want to write the output data to the Session Window.
5. Under **From** select the source of the data: a memory space, port, pin, register, a peripheral, history, or extended history.Selecting **History** writes a record to the file for each instruction execution. The last record in the file is always the next instruction to be executed. Selecting **Extended History** writes a record for each execution cycle. The Simulator logs additional execution history information to the output file, including device wait state cycles and bus arbitration cycles and indication of other stall conditions. The extra information is preceded by double asterisks in the log file.The output record contains a record number, the optional timing field containing the cycle number, and the data value. For the history file, the data is comprised of the pc (program counter), the instruction word(s) in hexadecimal, and the disassembled instruction.
6. If the output data is being written from a memory space, under **Memory** select the memory space. Then type in the **Address** referencing the output data. Assigning output to a memory address causes all subsequent writes of that memory address to write the data to the output file
7. If the output data is being sent from a port, under **Port** select the name of the port. The list of available ports varies depending on the device that you are targeting.To see a list of port names and their index and mask, type `help port` from the command line of the Command window .
8. If the output data is being sent from a pin, under **Pin** select the pin location.If the pin that you want to send the data from is a single location (as opposed to a range of pins) select the name of the pin under **Start**. If you want the output data to come from a range of pins, select the name of the **Start** and **End** pin. To see a list of pin names and their indices, type `help pin` from the command line of the Command window .
9. If the output data is being sent from a register, under **Register** select the register.
10. If the output data is being sent from a peripheral, under **Peripheral** select the name of the peripheral.The list of available peripherals varie depending on the device that you target. To see a list of peripheral names with indices, type `help periph` from the command line of the Command window .
11. Under **Radix** select the number system (hexadecimal, decimal, etc.) in which the output data should be written.
12. If the output data is being sent to a file, under **File Name** type in the name of the output file or click on the **File** button to browse for the file. The default filename extension

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

is `.io`.The Simulator can store data only or time-data pairs. The output time value is always expressed in decimal. The output file is in ASCII format.

13. Click **OK**.

Section 12.24, "output - Assign Output File," on page 12-33 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 6
Simulator and Device Configurations

6.1 Introduction to Simulator Configuration

You can specify a specific DSP configuration for each device. The selected device configuration determines the internal memory attributes and simulated peripherals. The Simulator determines external memory access by the device configuration.

To set the configuration of a device:

1. From the **Modify** menu select **Device**. Then choose **Configure**.
2. Under **Device** select the number of the device that you want to configure.
3. Under **Configure** select **Type**.
4. Under **Device Type** select the type of device that you want to simulate.
5. Click **OK**.

The Simulator contains the predefined memory map of the default device type as the default memory configuration. You can configure the device to exit its reset state in a pre-defined operating mode. Once the Simulator is active, you can change the operating mode under program control by changing the value of the device Operating Mode Register (OMR). The mode pins are automatically configured to the default operating mode. You can set the operating mode that the device simulates following a simulated hardware reset when resetting the device registers and memory

The full external memory map of the device is, by default, RAM memory. The large external memory space is simulated using a virtual memory technique which automatically pages memory blocks to disk if the operating environment fails to allocate the required space in memory.

By changing memory values, you can change the on-chip bootstrap and data ROM areas. You can specify the bootstrap ROM by using PR: as the memory space designator.You can specify the data ROMs by XR: or YR:. Loading an assembler output file with the Simulator load can also modify the bootstrap ROM or data ROM areas. You can reinitialize the ROM areas by selecting **Reset** from the **Execute** menu, then selecting **State**.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Section 2.9, "Resetting the Device," on page 2-7 provides more information.

6.2 Setting the Default DEVICE

You can specify a particular device as the default device.

To set the default device:

1. From the **Modify** menu choose **Device** then select **Set Default**. The following dialog box appears:

Figure 6-1. Set Default Device Dialog Box

2. Under **Device** select the number of the device that you are choosing as the default device (the current device).
3. Click **OK**.

The Simulator applies commands to the default device. When you change the default device the Session window and the Command window automatically change to reflect the information for the new device. However, other windows such as the Assembly window, Source window, etc. do not automatically update themselves to reflect the new device. This allows you to display the information for separate devices in separate windows. To see the information for the new default device, open another Assembly window, Source window, etc.

Section 12.8, "device - Multiple Device Simulation," on page 12-15 provides a complete command description.

6.3 Changing the Radix

You can change the default radix (number base) that the Simulator uses for command parameters and other values that you input. You can also change the radix used to display specific registers and memory locations.

As an alternative to changing the default radix, you can also denote the number base of a value by preceding it with a dollar sign ($), an apostrophe (’), or a percent sign (%).

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

$	value is hexadecimal
’	value is decimal
%	value is binary

The Simulator begins with the default radix set to decimal for input and hexadecimal for most output. Changing the default input radix allows you to enter values without having to type the radix specifiers before each value.

To change the radix of input:

1. From the **Modify** menu choose **Radix** then select **Set Default**. The following dialog box appears:

Figure 6-2. Set Default Radix Dialog Box

Note that the current default radix is automatically selected.

2. Under **Radix** select the number system to use. This system will be the default for values that you provide when performing commands.
3. Click **OK**.

6.3.1 Changing the radix in which to display a register or memory locations:

1. From the **Modify** menu choose **Radix** then select **Set Display**. The following dialog box appears:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

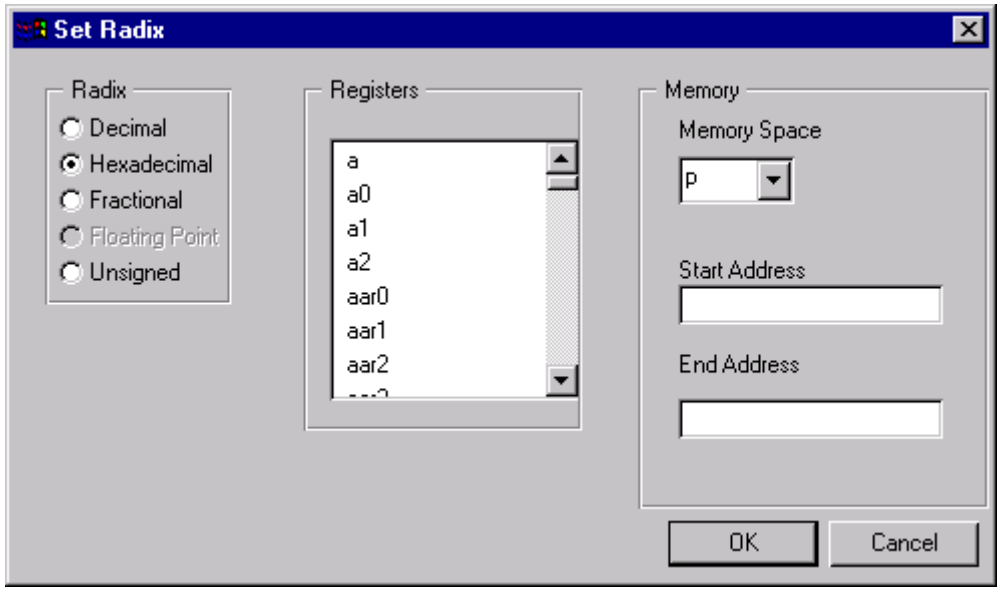

Figure 6-3. Set Register or Memory Radix Dialog Box

2. Under **Radix** select the number system to use.
3. Under **Registers** select the register that will be displayed with the new radix. To select more than one register, hold down the CTRL key while clicking once on the register names. This does not affect the default input radix.
4. Under **Memory** select the memory space and addresses that will be displayed with the new radix. This does not affect the default input radix.
5. Click **OK**.

Section 11.2, "Displaying the Radix," on page 11-2 provides more information about the radix; Section 12.27, "radix - Change Input or Display Radix," on page 12-36 provides a complete command description.

6.4 Loading Simulator State Files

You can load a previously saved simulation state file. The state file acts like an initialization file which includes information about:

- the state of all DSP devices in the system, and their device type
- enabled registers, counters, status registers, peripheral registers, etc.
- the entire contents of memory
- the windows settings
- the command history buffer
- the Session window output buffer for each device

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Essentially, a state file contains all the information needed to exactly duplicate the condition of the Simulator as it existed when it saved the state file.

You can use a state file in several ways. For example, if you are in a long development session and want to take a break, save the Simulator state to a state file so as not to lose your place. Or, if a particular part of a program is proving troublesome, you can save the state just before the problem area, simplifying the setup for repeated attempts to isolate the problem. There are, of course, many reasons for saving the state of the Simulator.

To load a Simulator state file:

1. From the **File** menu, choose **Load** then select **State**
2. Specify the name of the state file in the dialog box.
3. Click on **Open**. The current state of the Simulator is replaced by the state information in the .sim file.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

6.5 Changing and Saving Window Preferences

You can save the window positions when you exit the Simulator. Thus, when you restart the Simulator, it restores all windows to their positions on exit.

To save window status on exit:

1. From the **File** menu choose **Preferences**.

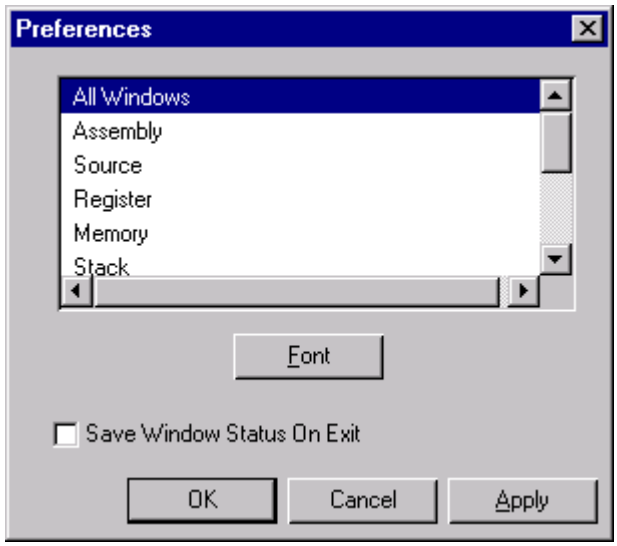

Figure 6-4. Window Preferences Dialog Box

2. Mark the checkbox labeled **Save Window Status On Exit**.
3. Click **OK**.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 7
Debugging C Source Code

7.1 Introduction to Debugging C Source Code

The Suite56 DSP Simulator provides several features specifically used in debugging C source code. The following topics will help you understand how best to use the Simulator for debugging C source code. (When you compile your source code, remember to include debug information.)

Section 12.1, "Command Overview," on page 12-1 also provides complete description of the commands discussed in this chapter.

7.2 Displaying the Call Stack

You can display the C function call stack beginning with the frame you specify.

To display the call stack:

1. From the **Display** menu choose **Call Stack**.
2. The following dialog box appears:

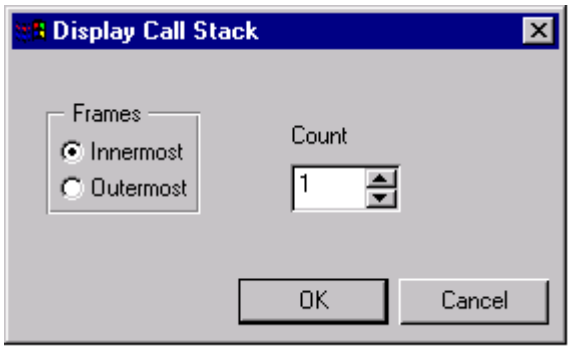

Figure 7-1. Displaying the Call Stack

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

3. Under **Frames**, select whether you want to display the innermost or outermost frames.
4. Under **Count**, specify the number of frames you want to display.
5. Click **OK**.

If you are writing a script, it can be useful to use the WHERE command to display the C function call stack.

Section 11.7, "Stack Window," on page 11-7 provides more information about the stack window. Section 12.46, "where - GUI C Calls Stack window," on page 12-51 provides a complete command description.

7.3 Moving Up and Down the Call Stack

You can move the current frame up and down the call stack.

To move up the call stack:

1. From the **Modify** menu choose **Up**. The following dialog box appears:

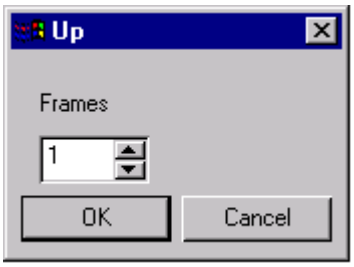

Figure 7-2. Moving Up the Call Stack

2. Under **Frames** select the number of frames to move up in the stack.
3. Click **OK**. The frame to which you moved is now the current starting point for evaluations.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

To move down the call stack:

1. From the **Modify** menu choose **Down**. The following dialog box appears:

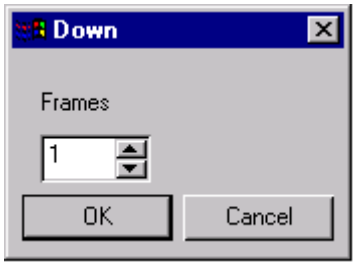

Figure 7-3. Moving Down the Call Stack

2. Under **Frames** select the number of frames to move down the stack.
3. Click **OK**. The frame to which you moved is now the current starting point for evaluations.

See Section 12.38, "up - Move Up the C Function Call Stack," on page 12-47 and Section 12.11, "down - Move Down the C Function Call Stack," on page 12-19 for complete command descriptions.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

7.4 Dynamically Displaying C Function Calls

You can monitor the calls that are made by C functions by displaying the Calls window. The Calls window dynamically monitors the C function calls as they occur.

To monitor C function calls:

1. From the **Windows** menu choose **Calls**.

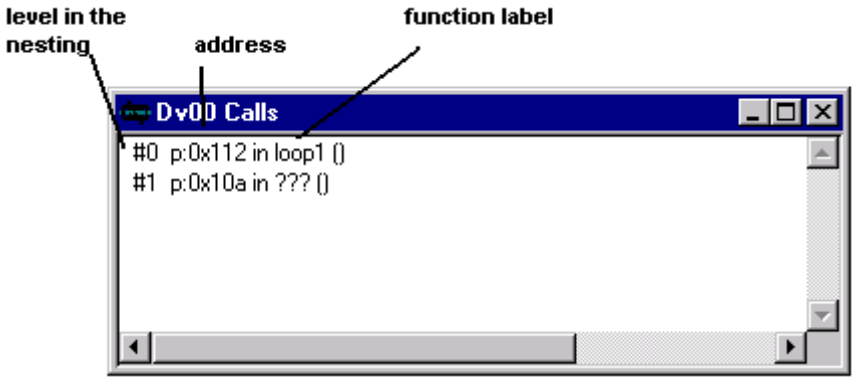

Figure 7-4. Dynamic Display of C Function Calls

   The Calls window updates as each instruction is executed. If the instruction contains no function call, three question marks are shown "???" to indicate that no function is recognized.

2. Double-click on a stack level to select it as the expression context to evaluate expressions.

Each function call adds another stack frame, each return removes one. Entry `#0` is the most nested function, that is the top entry on the stack. The highest number is the main function.

Each entry shows a number to indicate the level to which the call is nested, the PC return address (i.e. the address after the function call), and the name of the function. The top level represents the entry to the debug monitor, and so indicates the next instruction to be executed.

The call stack also indicates the context to use for evaluating C expressions. As each function may have its own copy of a named variable, it may be necessary to indicate which instance is required.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

7.5 Enabling/Disabling I/O Streams

You can enable or disable stream I/O for C programs that are running on the current device. The standard stream files are supported: stdin, stdout, and stderr. Any references by C programs to these files may be redirected to files on the host.

Stream file handling may be configured independently for each device. Streams handling is enabled by default. If a C program attempts to access a stream file while it is not enabled and redirected, the access is ignored. Output is discarded, and a standard value is supplied as input.

To enable or disable the stream I/O:

1. From the **File** menu choose **IO Streams**.
2. Select **Enable** or **Disable**.

Section 12.32, "streams - Enable/Disable Handling of I/O for C Programs," on page 12-42 provides a complete command description.

7.6 Redirecting an I/O Stream

You can redirect an I/O stream from the current device to a file on the host. Each stream file can be assigned individually; unwanted streams do not have to be redirected.

Streams can be redirected whether stream support is enabled or disabled; however, for the redirection to be effective, stream operations must be enabled. Disabling stream support while a stream is redirected does not terminate the redirection. It merely makes it ineffective until streams are enabled again.

To redirect the stream I/O:

1. From the **File** menu choose **IO Redirect** then select **Streams**.
2. Under **Streams** select the type of stream that you want redirected: `stdin`, `stdout`, or `stderr`.
3. Under **File Name** specify the filename to redirect the stream or click on the **File** button to browse for the file.
4. Click **OK**.

To stop redirecting the stream I/O:

1. From the **File** menu choose **IO Streams**, then select **Off**.
2. Select the type of stream to stop redirecting.
3. Click **OK**.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

To display redirected stream I/O:

1. From the **Display** menu choose **Redirected IO Streams**.
2. View the Session window. The redirected stream will be listed with the filename to which the streams are being directed.

Section 12.28, "redirect - Redirect stdin/stdout/stderr for C Programs," on page 12-38 provides a complete command description.

7.7 Display the Type of a C Variable or Expression

You can display the type of a C variable or expression. Keep in mind that you might need to move up or down the call stack to select the desired expression context. You can also change the current context when monitoring C function calls.

To display the type of a C variable or expression:

1. From the **Display** menu choose **Type**. The following dialog box appears:

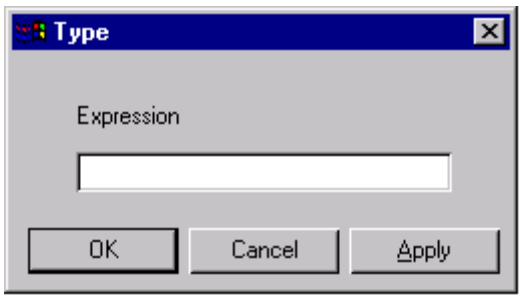

Figure 7-5. Displaying the Type of a C Variable or Expression

2. Type in the expression. Remember to enclose C expressions in curly brackets. For example:

   ```
   {gi * i}
   ```

3. Click **OK**.
4. The data type is displayed in the Session window. If the result of the expression is a storage location (for example, a variable name or an element of an array), the address of the storage location will be displayed in addition to its data type.

Section 9.1, "Introduction to Expressions," on page 9-1 provides more information about expressions; Section 12.35, "type - Display the Result Type of C Expression," on page 12-45 provides a complete command description.

Another useful tool for debugging C source code is the watch list. See Section 1.8, "Using a Watch List," on page 1-9 for step-by-step instructions on setting up a watch list.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Chapter 8
Macros, Scripts and Log Files

8.1 Creating and Running a Command Macro

The command macro is a useful tool for performing multiple Simulator commands. The command macro is useful for performing commonly repeated tasks such as setting the path, loading source programs, and setting up watch windows. However, a macro can consist of almost any Simulator command.

The command macro is a standard ASCII text file; you can create or edit it with any text editor. You can also create the macro by recording commands to a log file.

To record a command macro :

1. From the **File** menu, choose **Log**, then select **Commands**.
2. Specify the filename and save.The default filename extension is .cmd. A different extension can be used, for the sake of consistency we do not recommend this. If you specify a filename that already exists, you can choose to overwrite the file or append the new commands to the end of the existing file.
3. The log file is now recording all Simulator commands, whether issued from the menu bar or from the command line in the Command window. Use the Simulator commands just as you would during normal program execution. Note that there is an exception. If you stop program execution while recording commands, the stop is not recorded. The only way to pause execution from within a macro is to use STEP, NEXT, TRACE, UNTIL, WAIT or use a breakpoint. If you have any question about whether a command can be recorded, think of it this way: if it appears in the command window it is recorded.
4. To stop recording, from the **File** menu choose **Log**, then select **Close**.
5. From the dialog box select **Commands**.
6. Click **OK**. The command macro is now saved and can be replayed or edited.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

To run a command macro:

1. From the **File** menu, choose **Macro**.
2. Specify the file to run and click **Open**.The command macro begins. The commands are displayed in the Command window as they are executed. The commands are also echoed in the Session window, along with any output that is generated.

You can stop the command macro by choosing **Stop** from the **Execute** menu or by clicking on the **Stop Button** on the tool bar.

Section 2.6, "Pausing Execution with WAIT," on page 2-5 and Section 2.8, "Stopping Program Execution," on page 2-7 provide more information about stopping or pausing a command. Section 12.21, "log - Log Commands, Session, Profile," on page 12-30 provides a complete command description.

8.2 Logging Output from the Session Window

You can log all output that the Simulator displays in the Session window. This is especially useful if you are capturing information that spans several screens.

To log output displayed in the Session window:

1. From the **File** menu choose **Log**, then select **Session**.
2. Specify the filename that you want to give the log file and click **OK**. The Simulator writes all output that is echoed in the Session window to the log file you specified.

Section 12.21, "log - Log Commands, Session, Profile," on page 12-30 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 9
Expressions

9.1 Introduction to Expressions

The Simulator allows you to use an expression in most places where a constant is valid. For example, an expression can take the place of the start and stop location in the specification of an address range. An expression comprises a combination of symbols, constants, operators, and parentheses. Expressions follow the conventional rules of algebra and boolean arithmetic.

Expressions may contain any combination of integers, floating-point numbers, memory space symbols and register symbols.

9.2 Evaluate Expressions

You can evaluate DSP assembler expressions and C expressions and write the result to the session window.

To evaluate an expression:

1. From the **Evaluate** menu choose **Evaluate**. The following dialog box appears:

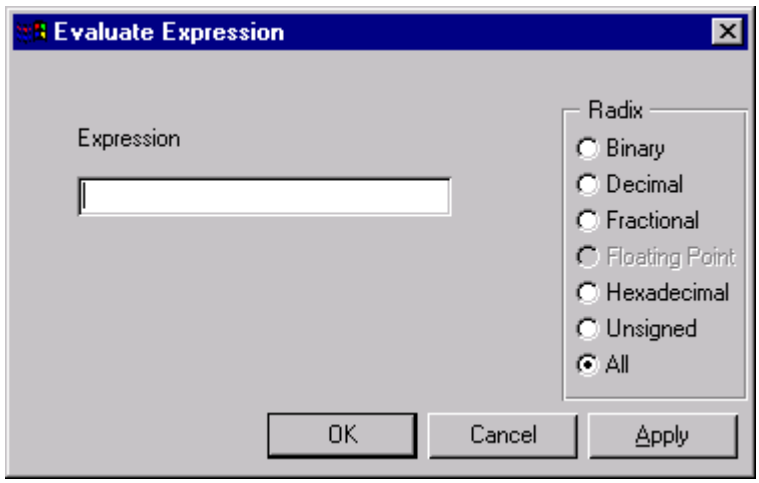

Figure 9-1. Evaluating an Expression

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

2. Under **Expression** type the expression that you want to evaluate If the expression is a C expression, enclose it in curly braces: {*c_expression*}
3. Under **Radix** select the radix in which to display the result of the expression. C expressions display the type of the expression and the value in normal format for the expression type if **All** is selected.Selecting **All** when evaluating a DSP assembler expression displays select radices depending on the expression itself.
4. Click **OK**. The Session window displays the result of your evaluation.

C expressions are evaluated in the context of the current stack frame by default – that is, the value displayed is that which would have been returned if the expression had been included in the program at the current execution point. C expressions can be evaluated in the context of any of the functions on the call path to the current function.

Section 12.12, "evaluate - Evaluate an Expression," on page 12-20 provides a complete command description.

9.3 Using Memory Space Symbols

The Simulator evaluator interprets a memory space symbol followed by an expression as the contents of a memory location. Use the Simulator's "`help mem`" command to obtain a list of the valid memory space prefixes and their corresponding address ranges. The following expression is converted to an integer constant in the address range of the DSP.

Some memory space symbols, such as `p:` or `x:`, require the evaluator to first read the device Operating Mode Register (OMR). Others, such as `xi:` or `pr:`, refer to an exact memory location regardless of the chip operating mode.

9.4 Using Register Name Symbols

The Simulator evaluator interprets a register symbol as the contents of that register. To obtain a list of valid register names use the Simulator "`help reg`" command on the command line in the Command window

Note that some hexadecimal constants, such as `a0` or `d1`, may also be valid register names for the selected DSP. It is usually best to precede such hexadecimal constants by a dollar sign **(**`$`**)** to distinguish them from registers of the same name.

Section 4.1, "Displaying Register Values," on page 4-1 provides more information on register values.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

9.5 Using Assembler Debug Symbols

The Simulator load command processes the symbol and line number information present in a COFF format object file (.cld file) which has been generated with the assembler's `-g` option. If symbol information has been loaded, the evaluator will accept symbol names or source file line numbers and translate them into an associated memory address.

In general you can reference a symbol name in the Simulator just as it was defined in the original source file, except that symbol names which conflict with a Simulator register name must be preceded by the `@` character. A symbol name may be further delimited by specifying a containing section name in the form `section_name@symbol_name`, with the `@` character being used as the separator. The section name global may be used for the global section. If a symbol is specified without a preceding section name, the evaluator assumes the section containing the current pc.

Line numbers may be expressed simply as a decimal integer preceded by the `@` character when referring to a line in the current source file. If an address field is being specified in a command, the line number's preceding `@` character may be omitted. A line number in a particular source file may be expressed in the form `source_filename@line_number`.

Below are valid forms of symbol names and line numbers:

Table 9-1. Valid Forms of Symbol Names and Line Numbers

Name Form	Explanation
`symbol_name`	translates to the address associated with symbol_name Example: `change pc lab_d`
`@symbol_name`	translates to the address associated with symbol_name Example: `disassemble @start_1`
`section_name@symbol_name`	translates to the address associated with symbol_name in section section_name Example: `display sec3@xdata`
`@section_name@symbol_name`	translates to the address associated with symbol_name in section section_name Example: `display @sec3@xdata`
`line_number`	translates to the address associated with line_number in the current source file. Example: `break 30`
`@line_number`	translates to the address associated with line_number in the current source file. Example: `change pc @30`

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 9-1. Valid Forms of Symbol Names and Line Numbers

Name Form	Explanation
`source_filename@line_number`	translates to the address associated with line_number in the named source file. Example: `change pc test.asm@30`
`@source_filename@line_number`	translates to the address associated with line_number in the named source file. Example: `change pc @test.asm@30`
`source_filename`	translates to the address associated with the first line in the named source file. Example: `list test.asm`
`@source_filename`	translates to the address associated with the first line in the named source file. Example: `list @test.asm`

9.6 Using Constants

Constants represent quantities of data that do not vary in value during the execution of a program.

9.7 Numeric Constants

Numeric constants can be in one of three bases:

- Binary constants consist of a percent sign (`%`) followed by a string of binary digits (`0,1`). For example:

  ```
  %11010
  %1001100
  ```

- Hexadecimal constants consist of a dollar sign (`$`) followed by a string of hexadecimal digits (`0-9,A-F or a-f`). For example:

  ```
  $FF
  $12FF
  $12ff
  ```

- Decimal constants can be either floating point or integer. Integer decimal constants consist of a string of decimal (0-9) digits. Floating-point constants are indicated either by a preceding, following, or included decimal point or by the presence of an upper or lower case '`E`' followed by the exponent. The special constants `inf` and `nan` can be used in floating-point expressions to represent the IEEE floating-point values of infinity and not-a-number for DSP devices which operate with IEEE floating-point values. For example:

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12345 (integer)

6E10 (floating point)

.6 (floating point)

2.7e2 (floating point)

A constant can be written without a leading radix indicator by changing the radix. For example, a hexadecimal constant can be written without the leading dollar sign ($) if the input radix is set to hex. The input radix defaults to decimal.

Section 6.3, "Changing the Radix," on page 6-2 provides more information about setting your radices.

9.8 Operators in Expressions

You can use some of the evaluator operators with both floating-point and integer values. If one of the operands of the operator has a floating-point value and the other has an integer value, the integer will be converted to a floating-point value before the operator is applied and the result will be floating-point. If both operands of the operator are integers, the result will be an integer value. Similarly, if both the operands are floating point, the result will be a floating-point value.

Operators recognized by the Assembler include the following:

Unary operators:

minus (-)

negate (~) - integer only

logical negate (!) - integer only

The unary negate operator will return the one’s complement of the following operand. The unary logical negation operator will return an integer `1` if the operand following it is `0` and will return a `0` otherwise. The operand must have an integer value.

Arithmetic operators:

addition (+)

subtraction (-)

multiplication (*)

division (/)

mod (%)

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

The divide operator applied to integer numbers produces a truncated integer result. The mod operator applied to integers will yield the remainder from the division of the first expression by the second. If the mod operator is used with floating-point operands, the mod operator will apply the following rules:

Y % Z = Y if Z = 0

= X if Z <> 0

where X has the same sign as Y, is less than Z, and satisfies the relationship:

Y = i * Z + X

where i is an integer.

Bitwise operators (binary):

AND (&) - Integer only

inclusive OR (|) - Integer only

exclusive OR (^) - Integer only

Bitwise operators cannot be applied to floating-point operands.

Shift operators (binary):

shift right (>>) - Integer only

shift left (<<) - Integer only

The shift right operator causes the left operand to be shifted to the right (and zero-filled) by the number of bits specified by the right operand. The shift left operator causes the left operand to be shifted to the left by the number of bits specified by the right operand. The sign bit will be replicated. Shift operators cannot be applied to floating-point operands.

Relational operators:

less than (<)

greater than (>)

equal (= =) or (=)

less than or equal (<=)

greater than or equal (>=)

not equal (!=)

Relational operators all work the same way. If the indicated condition is true, the result of the expression is an integer `1`. If it is false, the result of the expression is an integer `0`. For

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

example, if D has a value of 3 and E has a value of 5, then the result of the expression D<E is 1, and the result of the expression D>E is 0. Each operand of the conditional operators can be either floating point or integer. Test for equality involving floating-point values should be used with caution, since rounding errors could cause unexpected results.

Relational operators are primarily intended for use with the Simulator BREAK command.

Logical operators:

Logical AND (&&)

Logical OR (||)

The logical AND operator returns an integer 1 if both of its operands are non-zero; otherwise, it returns an integer 0.

The logical OR operator returns an integer 1 if either of its operands is non-zero; otherwise it returns an integer 0. The types of the operands may be either integer or floating point.

Logical operators are primarily intended for use with the Simulator BREAK command.

9.9 Operator Precedence

Expressions are evaluated with the following operator precedence:

1. parenthetical expression (innermost first)
2. unary minus, unary negate, unary logical negation
3. multiplication, division, mod
4. addition, subtraction
5. shift
6. less than, greater than, less or equal, greater or equal
7. equal, not equal
8. bitwise AND
9. bitwise EOR
10. bitwise OR
11. logical AND
12. logical OR

Operators of the same precedence are evaluated left to right. All integer results (including intermediate) of expression evaluation are 32-bit, truncated integers. Valid operands include numeric constants, memory addresses, or register symbols. The logical, bitwise,

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

unary negate, unary logical negation, and shift operators cannot be applied to floating-point operands. That is, if the evaluation of an expression (after operator precedence has been applied) results in a floating-point number on either side of any of these operators, an error will be generated.

9.10 Setting Up and Modifying a Watch List

You can watch the contents of a specific memory location, register, or any arbitrary value or expression by setting up a Watch window. Watch items are kept on a watch list that gets updated every time program execution is stopped.

The value or expression that you watch can be valid even if it is not calculated during program execution. C expressions can be used, enclosed in braces: `{c_expression}`. Symbolic references can be used if symbols have been loaded from the object module. The values are re-calculated and output at each break in execution.

To display a value in a WATCH list:

1. From the **Windows** menu choose **Watch**. The following dialog box appears:
2. Select the window number where you want the expression to appear. You will want to do this when you have more than one **Watch** window open.

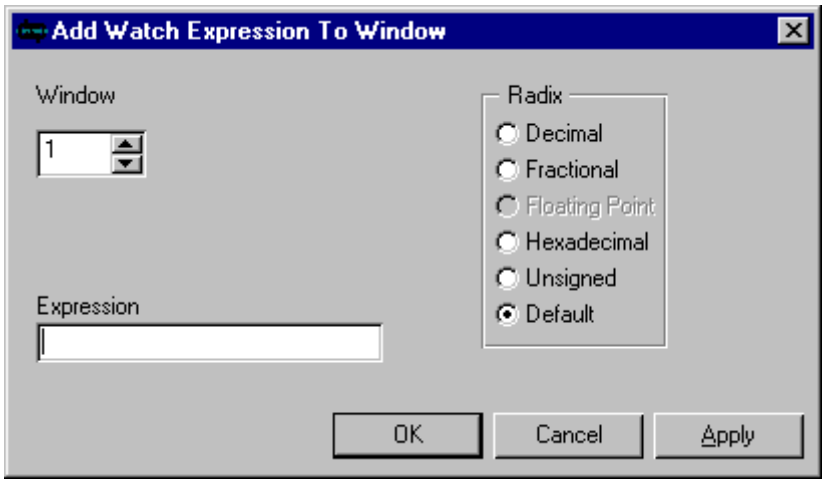

Figure 9-2. Displaying a Value in a Watch List

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

3. Under **Expression**, type in the expression that you want to appear in the Watch window.
4. Under **Radix**, select the radix format in which you want the variables displayed.
5. Click **OK**. The expression that you specified now appears in a Watch window. If the expression you type is not valid, you will get an error message explaining why the expression is not valid.

A C expression which refers to C variables can only be evaluated in the context in which the watch is established. That is, while all the variables used in the expression are in scope. So if one (or more) of the variables in an expression goes out of scope (either because a function call or return from a function), the value is replaced with the message **Expression out of scope**. When all elements of the expression are back in scope, the value is again displayed.

An expression which has gone out of scope because of function a call may be evaluated and displayed by selecting the stack frame for the evaluation context. The stack frame assignment remains in effect only until the next instruction is executed. An expression that is out of scope because of an exit from a function cannot be evaluated until the function is called again as its variables no longer exist.

Section 12.42, "watch - Set, Modify, View, or Clear Watch item," on page 12-49 provides a complete command description.

A WATCH list can also be used in conjunction with the EVALUATE command. See Section 9.2, "Evaluate Expressions," on page 9-1 for more information.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 10
Simulator Tool Bar

10.1 Using the Tool Bar

You can use the toolbar to control program execution. The icons located on the toolbar duplicate the same kind of control commands found under the **Execute** menu.

The icons on the toolbar are:

Go button
Starts program execution from the next address. All breakpoints are observed

Stop button
Stops program execution or a command macro, if running.

Step button
Executes next instruction or line.

Next button
Executes next instruction or line.

Finish button
Allows the current function to execute to completion.

Device button
Allows setting of the default device to which all commands will be directed.

Repeat button
Repeats the last command in the history buffer.

Reset button
Resets the current device.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

10.2 Go Button

The **Go** button starts program execution from the next address.

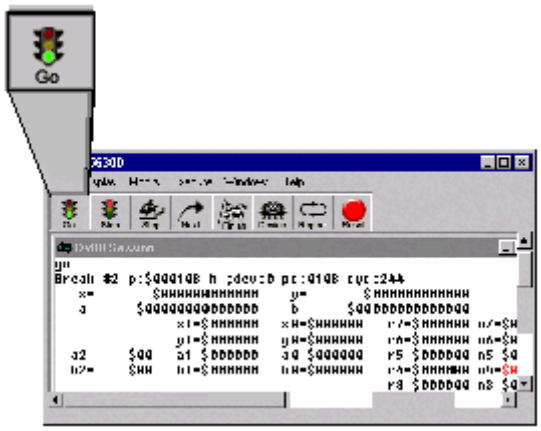

Figure 10-1. Go Button

Section 1.10, "Starting the Execution of Instructions with GO," on page 1-13 provides more information about starting program execution. Section 12.15, "go - Execute DSP Program," on page 12-22 provides a complete command description.

10.3 Stop Button

The **Stop** button stops program execution, or if a command macro is running, stops the macro.

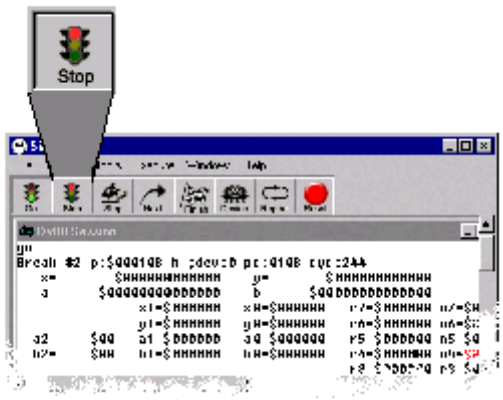

Figure 10-2. Stop Button

Section 2.8, "Stopping Program Execution," on page 2-7 provides more information about stopping program execution.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

10.4 Step Button

The **Step** button executes the next instruction or line.

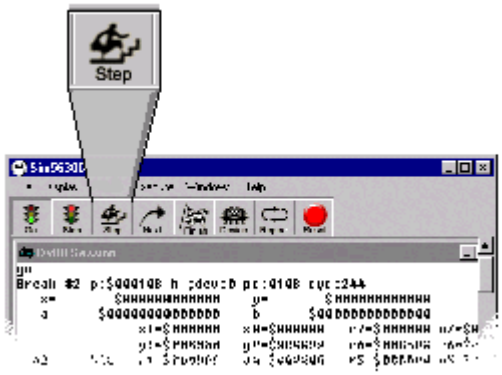

Figure 10-3. Step Button

If the Source window is open, displaying the program source, the **Step** button executes one line of code. Otherwise, the **Step** button executes one instruction. On encountering a JSR instruction, the Simulator begins with the first instruction of the function, and steps through it one instruction at a time.

To step one instruction at a time from the toolbar:

1. Make sure that program execution is halted and that the Source window is *not* open.
2. Click on the **Step** button on the toolbar.The Simulator executes the next assembly instruction.

To step one line at a time from the toolbar:

1. Make sure that program execution is halted and that the Source window is open. If the Source window is not open, the Simulator automatically executes the next instruction in the Assembly window rather than then next line in the Source window.
2. Click on the **Step** button on the toolbar. The Simulator executes the next executable line of code in the Source window. (Comment lines in the source code are ignored.)

Notice that the values in the Register window, the Memory window, and all other Simulator windows are updated each time you click on **Step**.

Section 2.2, "STEP Through Instructions," on page 2-1 and Section 2.4, "Executing the NEXT Instruction," on page 2-3 provide more information about stepping through instructions.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Section 12.31, "step - Step Through DSP Program," on page 12-41 provides a complete command description.

10.5 Next Button

The **Next** button executes the next instruction or line.

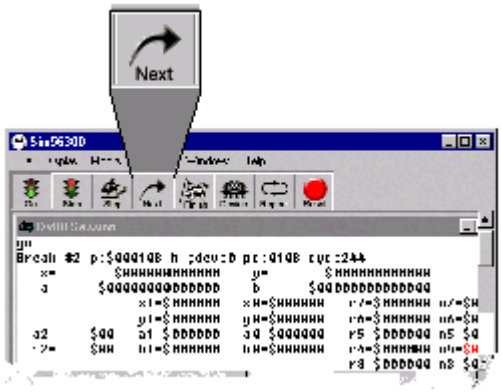

Figure 10-4. Next Button

If the next instruction to be executed calls a subroutine or begins execution of a function, all the instructions of the subroutine or function are executed before stopping. The enabled registers, memory, and other updated values are then displayed.

In order to recognize functions, the symbolic debug information for the program code must be loaded.

To execute the next instruction from the toolbar:

1. Make sure that program execution is halted and that the Source window is *not* open.
2. Click on the **Next** button on the toolbar. The Simulator executes the next assembly instruction.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

To execute the next line from the toolbar:

1. Make sure that program execution is halted and that the Source window is open. If the Source window is not open, the Simulator automatically executes the next instruction in the Assembly window rather than then next line in the Source window.
2. Click on the **Next** button on the toolbar. The Simulator executes the next executable line of code in the Source window. (Comments lines in the source code are ignored.)

Notice that the values in the Register window, the Memory window, and all other Simulator windows are updated each time you click on **Next**.

Section 12.23, "next- Step Over Subroutine Calls or Macros," on page 12-32 provides a complete command description.

10.6 Finish Button

The **Finish** button allows the current function to execute to completion.

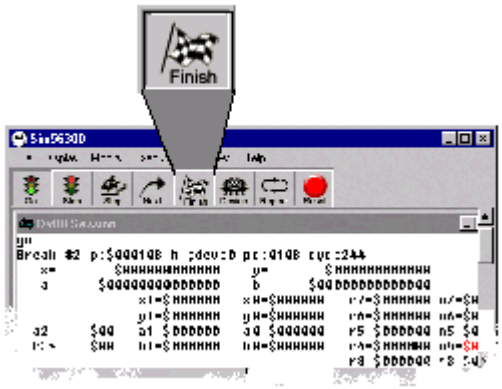

Figure 10-5. Finish Button

When stepping through a program, it is possible that execution will stop in the middle of a function or subroutine. Clicking on the **Finish** button completes the execution of the function or subroutine. The Simulator continues until encountering an RTS instruction. If another function is encountered during a finish operation, execution continues to the end of the current function.

Section 2.7, "Allowing the Current Function to FINISH after an UNTIL Condition or a Breakpoint," on page 2-6 provides more information. Section 12.13, "finish - Execute

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Until End of Current Subroutine," on page 12-21 provides a complete command description.

10.7 Device Button

The **Device** button allows you to set the default device to which all commands will be directed

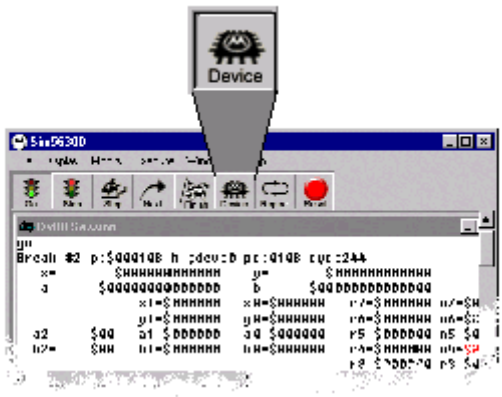

Figure 10-6. Device Button

Notice that when you designate another device as the default device that the Session window and Command window automatically reflect information for the new device. However, other windows such as the Assembly window, Source window, Stack window, etc. must be specifically opened to reflect the information for the new default device.

Section 6.2, "Setting the Default DEVICE," on page 6-2 provides more information about setting default devices. Section 12.8, "device - Multiple Device Simulation," on page 12-15 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

10.8 Repeat Button

The **Repeat** button repeats the last command in the history buffer.

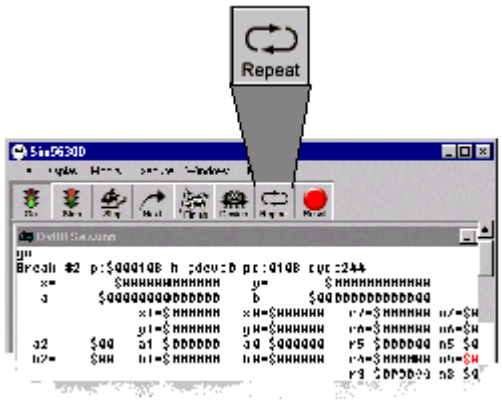

Figure 10-7. Repeat Button

The **Repeat** button repeats the last command in the history buffer - that is, listed in the Command window. Pressing this button is the same as clicking on the last command in the history buffer in the Command window and pressing <ENTER>.

Section 11.3, "Command Window," on page 11-2 provides more details about the Command window.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

10.9 Reset Button

The **Reset** button resets the current device.

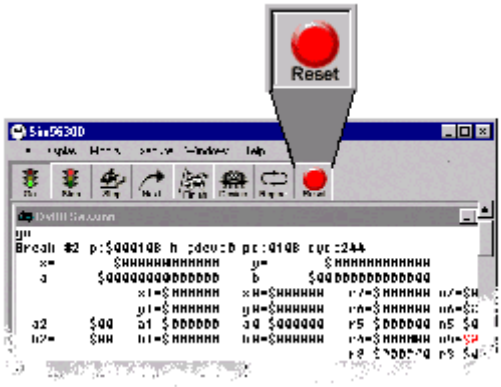

Figure 10-8. Reset Button

The **Reset** button generates a RESET command for the current device. The device will be reset in the same operating mode.

Section 2.9, "Resetting the Device," on page 2-7 provides more information about resetting devices. Section 12.29, "reset - Reset Device or State," on page 12-39 provides a complete command description.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 11
Displaying Information

11.1 Display the Current Breakpoints

It is often useful to display the enabled and disabled breakpoints for the current device by opening the Breakpoints window.

To display the current breakpoints:

1. From the **Windows** menu, choose **Breakpoints**. A dialog box shows a list of the breakpoints set for the current device. The following dialog box shows an example of a list of six breakpoints.

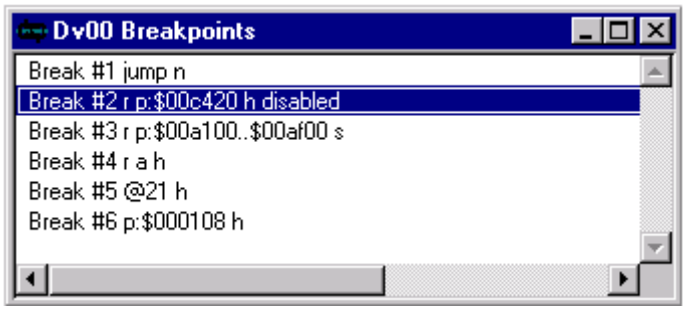
Dv00 Breakpoints
Break #1 jump n
Break #2 r p:$00c420 h disabled
Break #3 r p:$00a100..$00af00 s
Break #4 r a h
Break #5 @21 h
Break #6 p:$000108 h

Figure 11-1. Displaying the Current Breakpoints

   Disabled breakpoints such as the `Break #2`, are indicated with the word `disabled`.

2. Double click on a breakpoint from the list to toggle it from disabled to enabled, or from enabled to disabled.

Breakpoints that were set by double clicking on the Source window, such as `Break #5` above, are identified by the line number.

Breakpoints that were set by double clicking on the Assembly window, such as `Break #6` above, are identified by the address.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

11.2 Displaying the Radix

You can display the default radix (number base) that is currently used for command parameters and other values that you input. That is, the radix that is used when you type a value into a dialog box or at the command line. This is what is meant by the default radix.

To display the current default radix:

1. From the **Display** menu, choose **Radix**.
2. The default radix is displayed in the Session window.

If the default radix for the Simulator is set to decimal, this means that values can be entered in decimal without typing a preceding apostrophe (‘). If the default radix for the Simulator is set to hexadecimal, this means that values can be entered in hexadecimal without typing a preceding dollar ($). Similarly, if the default radix is unsigned or binary, their respective specifiers do not need to be typed before the value.

Be careful to distinguish the input radix from the output radix. The default radix refers to the input radix.

The output radix is the radix in which values are displayed. You can change the radix for a particular register or for a particular memory location. So be careful to note when you have changed the output radix of a particular register or memory location.

Section 6.3, "Changing the Radix," on page 6-2 provides more information about changing the radix.

11.3 Command Window

The Command window permits you to enter Simulator commands on a command line. In this way, the Command window provides an alternative to using the menu bar or the toolbar; the command line also let you monitor the syntax of the Simulator commands. The Command window is also a useful source of information in that it echoes the commands from the menu bar and from the toolbar. It also provides a source of history. The command history buffer holds the ten most recent commands. If the last command is repeated exactly, the duplicate is not stored.

To display the Command window:

1. From the **Windows** menu, choose **Command**. The Command window appears:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

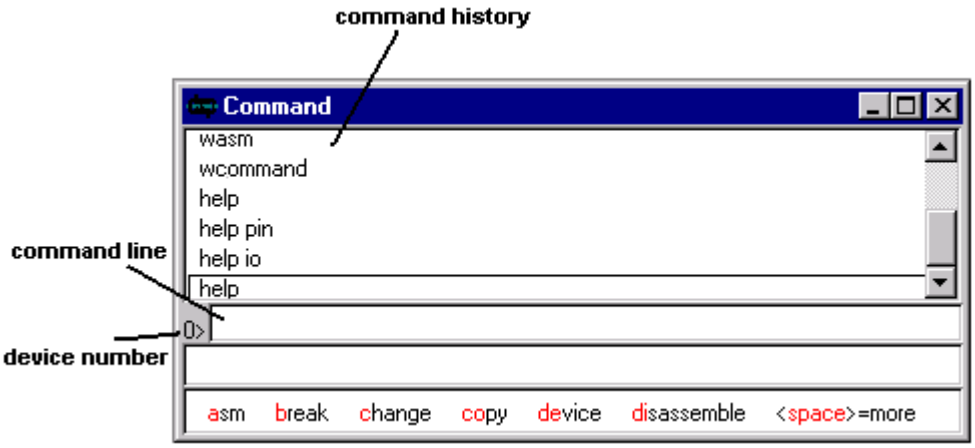

Figure 11-2. Displaying the Command Window

2. Notice that the device number is indicated. This is the device to which the commands are applied. In order to affect another device, the default device must be changed.

The Command window also provides device specific information that is not available elsewhere in the Simulator.

To display device specific information:

1. From the command line of the Command window, type:

   ```
   help
   ```

2. A list of help topics is displayed in the Session window. Check the list for the device specific information that you are wanting. You might have to scroll back up the Session window to see the whole list.
3. From the command line of the Command window, type help followed by the name of the help topic. For example, to list the on-chip I/O registers and their addresses, type:

   ```
   help io
   ```

   The help information is displayed in the Session window. Some of this information can be lengthy, so it might be useful to log the Session window output. See Section 8.2, "Logging Output from the Session Window," on page 8-2 for information about logging output.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

11.4 Session Window

The Session window is the main output for the Simulator. The Session window displays:

- an echo of all commands input at the command line (from the Command window)
- output from commands
- information from the Display menu
- views of source code and assembly code
- registers and memory locations enabled for display at breakpoints and after execution
- error messages

To display the Session window :

1. From the **Window** menu, choose **Session**.
2. The Session window opens. You will probably have to scroll and resize the window to be suitably visible. Note that it is not possible to have more than one Session window open at a time.

The Session window displays output for the current device only. However, each device writes output to its own buffer. That is, each device retains its own session buffer. In order to see the Session window for another device, you must specify the other device as the current device. When another device is selected as the current device, the Session window is refreshed with the buffer for that device.

The Session window buffers the last 100 lines of output. It might be necessary to scroll through the Session window to see previous output. At times, you might have a command that displays more than 100 lines of output to the Session window. In this case you will want to pause the output to the Session window.

To pause output to the Session window:

1. From the **Display** menu, choose **More**. Then select **On**.
2. The Session window will now pause if more than 100 lines of information are displayed at once. For example, if you perform a command such as typing `help io` from the command line, it is likely that the resulting output to the Session window will be greater than 100 lines. The following dialog box will appear:

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

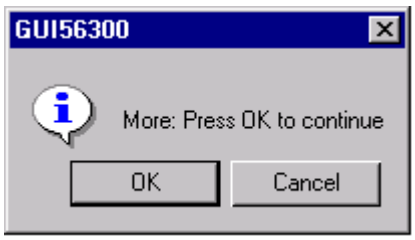

Figure 11-3. Pausing Output to the Session Window

3. Click on **OK** to display the next 100 lines in the Session window.

11.5 Assembly Window

The Assembly window allows you to display and edit the contents of memory, set and clear breakpoints, and follow program execution. As the program executes, the display is updated at each break in execution. The next instruction to be executed is always displayed, highlighted in red.

To use the Assembly window :

1. From the **Windows** menu, choose **Assembly**. The Assembly window appears:

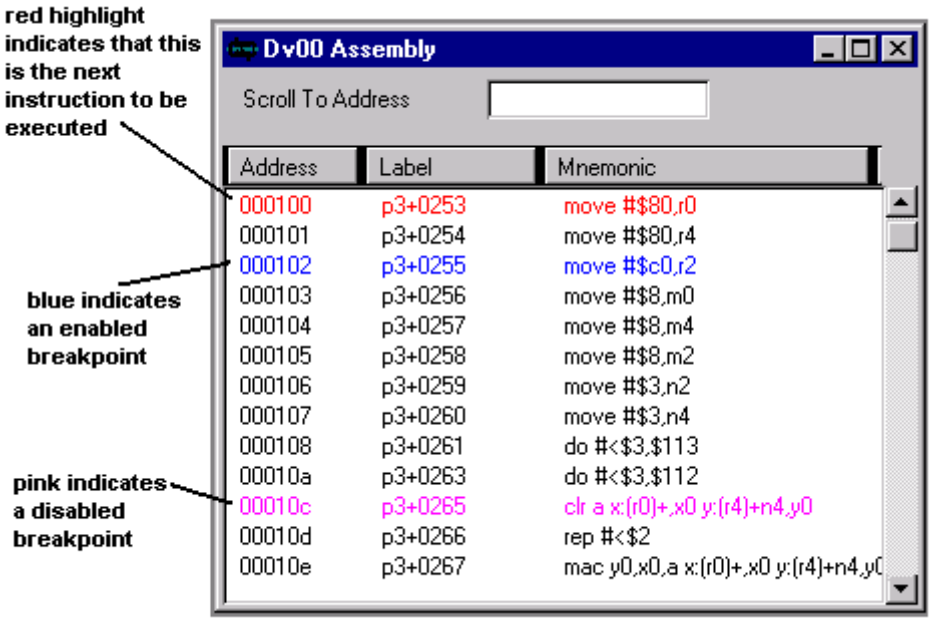
red highlight indicates that this is the next instruction to be executed
blue indicates an enabled breakpoint
pink indicates a disabled breakpoint
Dv00 Assembly
Scroll To Address
Address
Label
Mnemonic
000100 p3+0253 move #$80,r0
000101 p3+0254 move #$80,r4
000102 p3+0255 move #$c0,r2
000103 p3+0256 move #$8,m0
000104 p3+0257 move #$8,m4
000105 p3+0258 move #$8,m2
000106 p3+0259 move #$3,n2
000107 p3+0260 move #$3,n4
000108 p3+0261 do #<$3,$113
00010a p3+0263 do #<$3,$112
00010c p3+0265 clr a x:(r0)+,x0 y:(r4)+n4,y0
00010d p3+0266 rep #<$2
00010e p3+0267 mac y0,x0,a x:(r0)+,x0 y:(r4)+n4,y0

Figure 11-4. Using the Assembly Window

2. To scroll to a specific address, type the address in the box labeled **Scroll To Address** and press ENTER.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

3. To edit an instruction single click on the mnemonic. Notice that the mnemonic text is now highlighted. Type the new assembly instruction and press ENTER. Notice that the next instruction is highlighted. In order to avoid accidentally typing over that instruction click on an area outside of the mnemonic.
4. Breakpoints can be set, disabled, or cleared by double clicking on an address or label field.

Double click on the address or label to set the breakpoint. Double click again on the address or label to disable the breakpoint. Double click again on the address or label to remove the breakpoint.

Enabled breakpoints appear in blue. Disabled breakpoints appear in pink.

11.6 Source Window

The Source window displays the source code that has been loaded in memory. The source code may reside in the directory containing the object module or any or the directories specified in the path.

1. From the **Windows** menu, choose **Source** The Source window indicates the current program counter (PC) and highlights the corresponding source line red.

   To display the Source window :

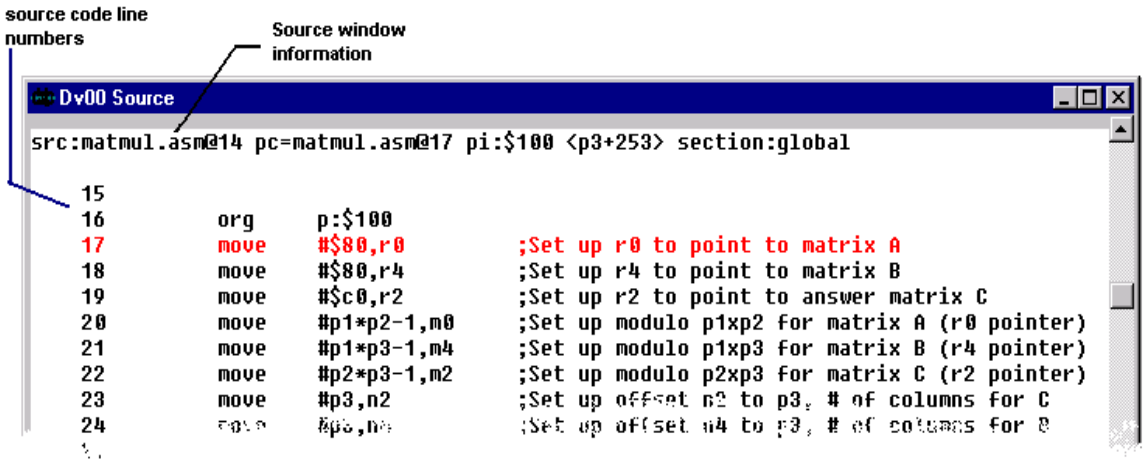
source code line numbers
Source window information
Dv00 Source
src:matmul.asm@14 pc=matmul.asm@17 pi:$100 <p3+253> section:global
15
16 org p:$100
17 move #$80,r0 ;Set up r0 to point to matrix A
18 move #$80,r4 ;Set up r4 to point to matrix B
19 move #$c0,r2 ;Set up r2 to point to answer matrix C
20 move #p1*p2-1,m0 ;Set up modulo p1xp2 for matrix A (r0 pointer)
21 move #p1*p3-1,m4 ;Set up modulo p1xp3 for matrix B (r4 pointer)
22 move #p2*p3-1,m2 ;Set up modulo p2xp3 for matrix C (r2 pointer)
23 move #p3,n2 ;Set up offset n2 to p3, # of columns for C

Figure 11-5. Displaying the Source Window

2. You can use the Source window to set halt breakpoints. To set or clear a breakpoint double-click on any line in the Source window.The breakpoint is added to the breakpoint list, displayed in the breakpoint window and highlighted in blue in the Assembly window.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

11.7 Stack Window

You can watch the hardware stack by opening the Stack window.

To display the hardware stack:

1. From the **Windows** menu, choose **Stack**. The hardware stack will be displayed.

Section 7.1, "Introduction to Debugging C Source Code," on page 7-1 provides more information about the Simulator’s hardware stack.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 12
Command Reference

12.1 Command Overview

There are a total of fifty-two Suite56 DSP Simulator commands that you can perform from the command line. (The command line is a part of the Command window.) These commands, and their syntax, are useful to know if you are writing a command macro.

The Simulator commands can be grouped into six categories:

- memory/register modification
- file I/O
- simulation execution control
- C source code debug commands
- miscellaneous tasks
- windows controls (used especially in writing command macros)

12.1.1 Memory/Register Modification

The following commands relate to modifying memory and registers. These allow you to:

- **assemble** (ASM) DSP instructions
- **change** register or memory locations
- **copy** a block of memory to a new location
- **disassemble** code stored in the simulated DSP memory space
- **display** registers and memory values
- **display** the Simulator revision number or memory configuration
- **reset** the device
- use the **history** command to disassemble and display the previous thirty-two instructions executed by the device
- use a **watch** list to display a variable whenever single stepping or program execution is halted

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.1.2 File I/O

The following commands relate to file input and output. These allow you to:

- **input** peripheral or memory location values from a file
- **output** peripheral or memory location values to a file
- **load** DSP Assembler object module files or previous simulation state files
- **log** Simulator commands, session display output or DSP program execution profile
- **save** Simulator memory to an object module file or the Simulator state to a state file

12.1.3 Simulation Execution Control

The following commands relate to control of program execution:

- specify **break** conditions
- **go** until a break condition is met
- **step** a specified number of instructions or cycles before displaying register and memory changes
- **trace** a specified number of instructions or cycles displaying register and memory changes at each step
- The **next** instruction operates essentially the same as the STEP instruction, except that if the instruction being executed calls a subroutine or function, execution continues until return from the subroutine or function.
- The **until** instruction has the effect of setting a temporary breakpoint at a specified address, executing until a breakpoint is encountered, then clearing the temporary breakpoint.
- The **finish** instruction executes instructions until the current subroutine or function completes.

12.1.4 C Source Code Debug Commands

The following commands relate to debugging C source code:

- **where** to display the C function call stack
- **up**, **down** and **frame** are used to traverse the call stack
- **redirect** is used to redirect data from stdin/stdout/stderr to files
- **streams** enable the io streams
- **type** displays the data type of a variable, function or C expression

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.1.5 Miscellaneous Tasks

The following commands include miscellaneous tasks:

- **device** specifies a new device and its device type
- **evaluate** evaluates expressions in five different radices
- **help** displays device specific information in the Session window
- **path** defines the working directory and alternate directories
- **quit** ends a simulation session
- **radix** specifies the default number base (hex, decimal, etc.) used during expression evaluation and data entry or data display
- **system** executes an operating system command in two modes
- **wait** specifies a number of seconds to pause a macro before proceeding to the next command
- **list** displays a specified source file when symbolic debug is in effect
- **view** allows selection of the Simulator display mode - Source, Assembly or Register
- **unlock** provides password enabling of unannounced device types

12.1.6 Windows Controls

The following commands control the display of windows and are particularly useful in writing command macros:

- **wasm** opens an assembly window displaying addresses, labels, and mnemonics
- **wbreakpoint** opens a breakpoint window displaying current breakpoints
- **wcalls** opens a C call stack window
- **wcommand** opens a command window
- **winput** opens an input window
- **wlist** opens a list window displaying contents of a specified file
- **wmemory** opens a memory window
- **woutput** opens an output window
- **wregister** opens a register window
- **wsession** opens the Session window
- **wsource** opens a Source window
- **wstack** opens a hardware stack window
- **wwatch** opens a watch window displaying expressions that have been specified for the watch list

For more information about macros, see section 8.1, “Creating and Running a Command Macro.”

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.2 Command Syntax

The command syntax line contains special punctuation to indicate command keywords, required or optional fields, repeated fields, and implied actions. For example, the syntax of the **display**, **evaluate**, **wait** commands looks like this:

DISPLAY [**ON/OFF/R/W/RW**] [reg[_block/_group]/addr[_block]]...

EVALUATE [**B**(binary)/ **D**(dec)/ **F**(float)/ **H**(hex)/ **U**(unsigned)] expression/**{**c_expression**}**

WAIT [count(seconds)]

The punctuation of the syntax line includes:

Table 12-1. Command Syntax Elements in Motorola DSP Documentation

Syntax	Syntax Description
CAPS	Capitalized words indicate command keywords. The user must enter command keywords exactly as shown (as opposed to variables or arguments that are provided by the user). Command keywords can be entered in either uppercase or lowercase in the software tool.
BOLD	In the command syntax line, boldface is only used in command keywords. The portion of the command keyword shown in boldface represents the minimum portion of the keyword that must be typed in order for the command to be valid. The portion of the keyword that is not in boldface can be typed if desired, but is not required. For example, in the **wait** command, the user only needs to type the letter w for the command to be recognized.
/	The slash is used to separate a list of mutually exclusive choices. The slash is not entered as a part of the command. For example, in the **evaluate** command, the first parameter can be one, and only one, of the following: B, D, F, H, or U.
( )	Parentheses surround a description of an implied action. This is included to help clarify some of the abbreviations used in the command syntax line. Neither the parentheses nor the description within are entered as part of the command. For example, when entering the **evaluate** command, the optional first parameter consists of one letter. The words binary, dec, float, hex, and unsigned are only included in the command syntax line for clarity.
...	An ellipsis (three consecutive periods) indicates that a parameter can be repeated several times in a command line. For example, the **display** command can specify multiple registers (or register blocks).
[ ]	Square brackets indicate enclose command parameters that are optional. The brackets themselves are not entered as a part of the command. For example, in the **wait** command the count parameter is optional.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.3 Command Parameters: List of Abbreviations

Many of the command parameters, such as `addr` and `reg`, are abbreviations. The following is a list of abbreviations used for parameters on the command syntax line:

Table 12-2. Command Parameter Abbreviations

Abbreviation	Meaning
`addr`	An address may be specified as a source file line number or as a symbol name if a previously loaded COFF object file contains symbolic debug information. Otherwise, a memory space designator must be used. Use the `help mem` **c**ommand from the command line of the Command window to obtain a list of the valid memory space prefixes.
`addr_block`	addr**..**location/addr**#**count
`bn`	Break number. A decimal integer constant in the range 1 to 99.
`break_action`	**H**(halt)/**I**n(increment CNTn)/**N**(note)/**S**(show)/**X** [command]
`count`	Positive integer expression in range 1 to $7fffffff.
`dev_num`	Device number in the form `DVn`, where `n` represents the number of the device. Range is `dv0` to `dv31`.
`dev_type`	Device type.
`expression`	Any arithmetic expression valid for the DSP Assembler. Register names can also be used in the expression.
`c_expression`	Any expression valid in the current C program. A `c_expression` must be enclosed in curly braces: { }
`file`	Any valid filename, including relative and absolute paths.
`ioradix`	**-RD**(decimal)/**-RF**(float or fractional)/**-RH**(hexadecimal)/**-RU**(unsigned)
`location`	Integer expression. It will be mapped into the device address range. For ex-ample, -1 translates to the maximum address.
`mode`	Device operating mode in the form **Mn**.
`pathname`	Any valid pathname.
`periph`	Valid peripheral names are displayed by the Simulator `help periph` command.
`pin`	Valid pin names are displayed by the Simulator `help pin` command. A pin name may optionally be preceded by `pin:` in order to resolve conflicts that may exist between pin and register names or constants.
`pin_block`	pin**..**pin
`port`	Valid port names are displayed by the Simulator `help port` command

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Table 12-2. Command Parameter Abbreviations (Continued)

Abbreviation	Meaning
`reg`	Valid register names are displayed by the Simulator `display all` command. A register name may optionally be preceded by `reg:` in order to resolve conflicts that may exist between register and pin names or constants.
`reg_block`	reg..reg
`reg_group`	An address may be specified as a source file line number or as a symbol name if a previously loaded COFF object file contains symbolic debug information. Otherwise, a memory space designator must be used. Use the `help mem` **c**ommand from the command line of the Command window to obtain a list of the valid memory space prefixes.
`topic`	addr**..**location/addr**#**count

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.4 asm - Single Line Interactive Assembler

ASM [**B**(byte wide)] [(beginning at)addr] [assembler_mnemonic]

The **asm** command allows you to create or edit DSP object code in memory using assembly language mnemonics. The assembler mnemonic is immediately converted into machine language code and stored in memory. The line of assembly source code is not saved.

[**B**(byte wide)] — Optional. When provided, this parameter causes the `assembler_mnemonic` to be divided and stored as a sequence of single bytes. This option is typically used to store a program as a byte-wide program in the bootstrap ROM area of the target device. This option could also be used to store a byte-wide program in external memory, which might hold, for example, overlay program code.

[(beginning at)addr] — Optional. The addr parameter specifies the beginning address to be edited. The address can be in any of the memory maps (p, x, y, pr, etc.) of the target device. (The Y memory is only valid for DSP56000 and DSP96002 family members. Use the Simulator's **help mem** command to obtain a list of the valid memory space prefixes.)

If no address is specified, assembly begins in the p (program) memory space using the current program counter value as the beginning address. The pr memory designation specifies the special bootstrap ROM area of the DSP.

[assembler_mnemonic] — Optional. The mnemonic to be written to memory. If no `assembler_mnemonic` is specified on the command line, an interactive mode of the **asm** command will be initiated. Invoking the intereactive mode causes the object code at the beginning address to be disassembled and displayed on the screen. You can then edit the instruction. If the new instruction cannot be assembled correctly an error message is displayed on the error line of the command window and the cursor is placed at the point of error. Pressing the ESC key causes the interactive **asm** command to terminate.

Table 12-3. asm Commands

Command	Resulting Simulator Action
`asm p:$50`	Starts the interactive assembler at address 50 (hex) in p memory
`asm x:0 move r0,r1`	Writes the instruction `move r0,r1` to address 0 in the x memory space.
`asm`	Starts the interactive assembler at the current program counter value.
`asm lab_d+5`	Starts the interactive assembler at symbolic address lab_d+5.
`asm myfile.asm@7`	Starts the interactive assembler at the address corresponding to myfile.asm line 7.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 12-3. asm Commands

Command	Resulting Simulator Action
`asm b y:$040100`	Performs byte-wide assembly from address $40100 in the pr memory space. Each byte of the instruction is stored in successive locations, so that two or three locations are required to store each 16- or 24-bit instruction. If assembled into program memory, this code cannot be executed directly; it is intended for use with code similar to the byte-wide loader in the ROM bootstrap code. Byte-wide assembly may be used interactively (as in this example) or to assemble a single instruction.

12.5 break - Set, Modify, or Clear Breakpoint

BREAK [#bn] [expression] [break_action]

BREAK [#bn] **R**(read)/**W**(write)/**RW**(access) reg/addr[_block] [break_action]

BREAK [#bn[,bn,...]] **OFF**/**E**(enable)/**D**(disable)

BREAK [#bn] **DR**(dma read)/**DW**(write)/**DRW**(access) addr[_block] [break_action]

The **break** command can be used to set, modify, or clear a breakpoint condition and to specify the action that occurs if the breakpoint condition is true. The **break** command has four possible forms as indicated by the four command syntax lines above. The first form causes a break condition if the evaluated expression is non-zero. The second form causes a break condition if a selected register or memory location is accessed by the core. The third form permits a breakpoint, or list of breakpoints, to be selectively enabled, disabled or deleted. The fourth form causes a break condition if a selected memory location is accessed by a DSP DMA (Direct Memory Access) controller. It is valid only for devices with on-chip DMA controllers.

`[#bn]` — Optional. Break number. The break number is used to assign a number to a breakpoint definition or to specify an existing breakpoint. A break number can be any positive decimal integer in the range 1 to 99. If you do not specify a breakpoint number, the Simulator automatically assigns the lowest unused number.

`[expression]` — Optional. A breakpoint expression can be any logical expression that is valid for the DSP Assembler. If the .cld file contains C symbolic debug information, breakpoint expressions can include any valid C expression for the program. (Remember to enclose C expressions in curly brackets.)

Another useful form of a breakpoint expression breaks at an address only when the opcode from that memory location is being decoded for next cycle execution. Other forms of the breakpoint expressions that check the value of the pc register or check for a read of a p memory location are less definitive due to the pipelined prefetch of the device. This special form of breakpoint is indicated by specifying the breakpoint expression as a single address in p memory.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

The following is a list of operators that can be used in the breakpoint expression:

Table 12-4. Breakpoint Operators

Operator	Operator Function
<	less than
&&	logical "and"
<=	less than or equal to
||	logical "or"
==	equal to
!	logical "negate"
>=	greater than or equal to
&	bitwise "and"
>	greater than
|	bitwise "or"
!=	not equal to
~	bitwise one's complement
+	addition
^	bitwise "exclusive or"
-	subtraction
<<	shift left
/	division
>>	shift right

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

See Chapter 5, "Device I/O and Peripheral Simulation," on page 5-1 for more detailed information on expression evaluation.

The breakpoint expression usually involves comparison of register or memory values. Any register name may be used in an expression. There are also two special flag variables that may be referenced in the breakpoint expression:

Table 12-5. Flag Variables in a Breakpoint Expression

Variable	Expression Returns
`eof`	True if an end-of-file condition occurs in an input file assigned to a peripheral or memory location.
`jump`	True if a "jump" change of flow occurs during code execution.

`[break_action]` Optional. The Simulator can take various actions when a breakpoint is encountered during DSP program execution. If no `break_action` parameter is entered, the default action is to halt program simulation and to display all enabled registers and memory blocks. Alternative Simulator actions can be specified by entering one of the following `break_action` parameters:

Table 12-6. Break_Action Parameters

Parameter	Resulting Simulator Action
H	Halts execution. This is the default when no `break_action` is specified.
In	Increments the `n` counter register (CNT`n`), where `n` represents the number of the counter to increment. There are four possible counters (n=1..4).
N	Note - displays the breakpoint expression in the Session window and continues with execution.
S	Show - displays the enabled register/memory set in the Session window and continues with execution.
X	Executes a Simulator command at the breakpoint. The command can be any valid Simulator command, except for commands that are related to device execution. Device execution commands, such as step, trace or go, will not execute.

R(read)/**W**(write)/**RW**(access)

Used with the second form of the **break** command. Indicates the type of access (read, write, or both) of a register or address that should be detected in order for the breakpoint condition to be true.

`reg/addr[_block]` Register or address (or address block). A range of addresses can be specified in two ways. One way is to type the beginning address followed by two periods and the ending adress. For example:

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

`p:$101..p:$107`. Another way is to type the beginning address followed by the pound sign and the size of the range. For example, to specify a range from address p:$101 to p:$107 type: `p:$100#7.`

If the .cld file contains symbolic debug line number information, breakpoint addresses may also be specified by using line numbers in source code that correspond to an address in memory.

`[#bn[,bn,...]]` Optional. Break number(s). Used with the third form of the **break** command to specify one or more break numbers. Each break number should be separated by a comma. Consecutive numbers can be represented by two periods. For example, to indicate break numbers 3 through 8, type: `#3..8`

OFF/E(enable)/**D**(disable)

Used with the third form of the **break** command. This form of the beak command is used to indicate that the breakpoint should be removed (**OFF**), enabled (**E**), or disabled (**D**).

DR(dma read)/**DW**(write)/**DRW**(access)

Used with the fourth form of the **break** command. Indicates the type of access by a DSP DMA (Direct Memory Access) controller that should be detected for the breakpoint condition to be true.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Table 12-7. BREAK Commands

Command	Resulting Simulator Action
`break`	Display all currently enabled breakpoints.
`break off`	Remove all breakpoints.
`break #1,3,5..9 off`	Remove breakpoints numbers 1, 3 and 5 through 9.
`break pc>=$500`	Halt DSP program simulation and display enabled registers and memory when the program counter register is greater than or equal to hexadecimal 500.
`break (lc<10)&&(pc>100)`	Halt if the loop counter is less than 10 and the program counter is greater than 100.
`break jump n`	Display breakpoint message if a jump change of flow occurs during execution.
`break eof||pc>$fff`	Halt if an end of file condition occurs in an assigned peripheral input file or if the program counter is greater that hexadecimal FFF.
`break r0==r1`	Halt when the value of register r0 equals the value of register r1.
`break lc>0&&jump i1`	Increment variable cnt1 if a jump occurs and the loop counter is greater than 0.
`break r r0`	Halt if register r0 is accessed for a read operation.
`break p:100`	Halt if the execution address is p:100.
`break w lc`	Halt if the loop counter register is written during code execution.
`break 10 x evaluate h r0`	Set a breakpoint at the address corresponding to line 10 of the current source file. Execute the Simulator command "evaluate h r0" when the breakpoint occurs.
`break myfile.asm@20`	Set a breakpoint at the address corresponding to line 20 of source file myfile.asm.
`break r xdat..xdat+50`	Halt if a read occurs from one of the 50 addresses beginning at the address associated with the symbol xdat.
`break rw p:30..40 s`	Display enabled registers and memory and continue program simulation if any program memory location from decimal 30 to 40 is accessed.
`break #1..10 d`	Disable breakpoints numbers 1 through 10.
`break {j==2}`	Halt if the C expression "j==2" is true.
`break e`	Enable all breakpoints.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.6 change - Change Register or Memory Value

CHANGE [register[_block]/addr[_block] [expression]]...

The **change** command can be used to change the value of a register, memory location, block of registers, or block of memory.

[register[_block]/addr[_block]

Optional. The register(s) or adress(es) to be changed. A register block is represented by two register names separated by two periods. For example, `r0..r3` means registers r0 through r3. A memory block can be specified using `start_addr#count,` where `count` represents the size of the block. An address block can also be specified with `start_addr..end_addr`. For example: `p:$5#20` indicates 20 locations beginning from program memory location 5; `p:5..24` indicates program memory locations 5 through 24.

[expression]] Optional. The expression to write to the specified register or address. The expression can be a simple constant value or a complex expression with multiple operators and operands. A more extensive discussion of valid expressions is presented in Chapter 5, "Device I/O and Peripheral Simulation," on page 5-1.

Multiple register names, memory locations and expressions can be specified on the same command line. Each specified destination must be followed by the value or expression to be assigned to it.

An interactive mode of the **change** command can be initiated by specifying a single register or memory location without an associated expression. In this mode each register or memory location can be examined and optionally modified. Typing the ESC key causes the interactive mode of the **change** command to terminate.

Table 12-8. CHANGE Commands

Command	Resulting Simulator Action
`change pc`	Displays register values individually starting with the program counter and prompts you for new values.
`change xi:$55`	Displays internal x memory location hexadecimal 55 and prompts you for a new value.
`change p:$20 $123456`	Changes the value at p memory address hexadecimal 20 to hexadecimal 123456.
`change xdat $234`	Changes the value at the x memory address that corresponds to symbolic label xdat to hexadecimal 234.
`change xdat..xdat+5 35`	Changes the values in the memory block beginning at the address corresponding to symbolic label xdat and ending at xdat+5 to decimal value 35.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 12-8. CHANGE Commands (Continued)

Command	Resulting Simulator Action
`change r0..r3 0 pi:$30..$300 0 x:$fffe $55 pc 100`	Changes the values in registers r0 through r3 to 0, internal p memory addresses hexadecimal 30 through 300 to 0, x memory address hex fffe to hex 55 and the program counter to decimal 100.

12.7 copy - Copy a Memory Block

COPY (from)addr[_block] (to)addr

The **copy** command copies memory blocks from one location to another. The source and destination memory maps may be different. This allows you to move data or program code from one memory map to another or to a different address within the same memory map.

`(from)addr[_block]` The memory address or memory address block from which to copy. A memory block can be specified using `start_addr#count,` where `count` represents the size of the block. An address block can also be specified with `start_addr..end_addr`. For example: `p:$5#20` indicates 20 locations beginning from program memory location 5; `p:5..24` indicates program memory locations 5 through 24.

`(to)addr` The memory address to which the data will be copied. If an `addr_block` was specified, this address will be the beginning address to which the data will be copied.

Table 12-9. COPY Commands

`copy pi:$100..$500 x:$500`	Copies the values located in internal program memory, addresses hexadecimal 100 through hexadecimal 500 to x memory starting at address hexadecimal 500.
`copy x:0#100 p:0`	Copies one hundred memory locations beginning at x memory address 0 to p memory beginning at address 0.
`copy lab_1#100 lab_2`	Copies one hundred memory locations beginning at the memory location corresponding to symbolic label lab_1 to memory beginning at the address corresponding to symbolic label lab_2.
`copy xdat..xdat+40 ydat`	Copies 40 memory locations beginning at the address corresponding to symbolic label xdat to the block beginning at the address corresponding to symbolic label ydat.
`copy p:0..20 p:40`	Copies p memory addresses 0 through 20 to p memory addresses 40 through 60.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.8 device - Multiple Device Simulation

DEVICE [**DV**n [dev_type/**ON/OFF/X**]]

The **device** command allows you to create and manage simulated target systems consisting of multiple DSP devices. It allows you to add a simulated device on which commands can be executed, to disable or enable existing devices, or to delete a device. If the **device** command is typed with no parameters, a list of all the possible device numbers that you can assign will be displayed including each device's status and device type.

DVn — Optional. Device munber. The device number is a number assigned to a particular device. The number is in the form DVn, where n is an integer from 0 to 31. (A target system can be comprised of up to 32 devices.)

If the **device** command is executed with a device number and no dev_ type, the device number specified will become the default device. If the device does not yet exist, it will be created with the default device type and made active.

[dev_type] — Optional. Specifies the device structure to be simulated by device DVn. Non-disclosed devices must be unlocked with the **unlock** command prior to using the **device** command.

ON — Optional. Enables the specified device (DVn). This means that the device will respond to commands that cause device execution (**go**, **step** or **trace**). During execution cycles, each enabled device executes a single clock cycle in turn. Device to device pin interconnections specified by the input command are updated following each cycle for active devices.

OFF — Optional. Disables the specified device (DVn). Disabling a device causes the device to not respond to commands that cause device execution (**go**, **step** or **trace**).

X — Optional. Deletes device DVn. If device DVn is the current device, it will be deleted and another device will become the current device. The Simulator requires at least one device to be allocated, which means that it is not possible to delete the current device if it is the only device in the target system.

Table 12-10. DEVICE Command

Command	Resulting Simulator Action
device	Displays a list of all devices and their current status. Also displays the list of possible device types.
device dv9	Makes device dv9 the current device. If device dv9 does not exist, creates device dv9. Because no device type is specified, it is initialized with the default device type.
device dv1 on	Enables device dv1 cycle execution. If device dv1 doesn't exist, creates it with the default device type.
device dv0 56116	Creates device dv0 and initializes it as device type 56116.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.9 disassemble - Object Code Disassembler

DISASSEMBLE [**B**(byte wide)] [addr[_block]]

The **disassemble** command allows you to review DSP object code in its machine language and assembly language mnemonic format. Invalid opcodes are disassembled to a define constant (DC) mnemonic.

[B(byte-wide)] — Optional. Constructs the instruction words by taking one byte from each word of memory, starting from the specified address.

[addr[_block] — Optional. Specifies the address or range of addresses to disassemble. (Symbolic labels, and line numbers in source code that correspond to an address in memory can also be used.)

A range of addresses can be specified by typing `start_addr#count`, where `start_addr` is the first address of the block and `count` represents the size of the block. An address block can also be specified by typing `start_addr..end_addr`. For example: `p:5#20` indicates 20 locations beginning from program memory location 5; `p:5..24` indicates program memory locations 5 through 24.

If no address is specified, the next 20 instructions will be disassembled, beginning with the address indicated by the value in the program counter.

Table 12-11. DISASSEMBLE Commands

Command	Resulting Simulator Action
`disassemble`	Disassemble the next 20 instructions beginning with instruction pointed to by the program counter. Repeatedly entering this command will result in consecutive 20 instruction blocks being disassembled.
`disassemble pr:0..20`	Disassembleprogram bootstrap ROM memory address block 0 to 20.
`disassemble lab_1..lab_2`	Disassemble memory address block beginning at the address corresponding to symbolic label **lab_1** and ending at **lab_2**.
`disassemble xdat#20`	Disassemble10 instructions beginning at the address corresponding to symbolic label **xdat**.
`disassemble 7`	Disassemble instructions beginning at the address corresponding to line 7 in the current source file.
`disassemble test.asm@8`	Disassemble instructions beginning at the address corresponding to line 8 in the source file **test.asm**.
`disassemble x:$50#10`	Disassemble10 instructions starting at x memory map hex 50.
`disassemble b y:$1000#$40`	Disassemble40 instructions starting at address y:$1000. The instruction words are constructed by taking one byte from each location; thus depending on the target processor, two or three locations are required to hold each instruction word.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.10 display - Display Register or Memory

DISPLAY [**ON/OFF/R/W/RW**] [reg[_block/_group]/addr[_block]]...

DISPLAY **V**(version)

The **display** command allows you to examine the contents of a specific register, a register group or a memory block. It can also be used to enable, conditionally enable, or disable the particular registers or memory locations that are displayd when a debug command such as **step** or **trace** is executed. You can also display the Simulator version number. Entering the **display** command with no parameters will display all enabled registers and memory blocks in the Session window.

The default radix for the display of most registers and memory locations is hexadecimal. However, you can specify the display radix of each register and memory location individually by using the **radix** command.

[ON/OFF/R/W/RW] Registers and memory locations can be enabled or disabled by entering the command with one of these keywords:

Table 12-12. DISPLAY Enable Keywords

Enable Keyword	Resulting Simulator Action
ON	Enables the specified registers and memory locations so that they are always automatically displayed when program execution is stopped. If no registers or memory locations are specified, all registers are enabled for display.
OFF	Disables the specified registers and memory locations so that they are not automatically displayed when program execution is stopped. If no registers or memory locations are specified, all registers and memory locations are disabled for display.
R	Displays the following registers and memory locations if they were accessed for a read operation since the last display.
W	Displays the following register and memory locations if they were accessed for a write operation since the last display.
RW	Displays the following register and memory locations if they were accessed for read or write operations since the last display.

The R, W, and RW functions initiate the accumulation of a list of accesses from display to display. All accesses to register locations can be saved. The memory lists store a maximum of 16 memory accesses (for each memory space). If more than 16 locations were accessed since the previous display, only the last 16 will be stored. Register and memory locations that have been accessed for a write operation are highlighted when displayed.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

`[reg[_block/_group]/addr[_block]]...`

Optional. The register(s) or address(es) to display. Entering the **display** command with a register or address parameter, but without one of the "enable" keywords, will cause immediate display of the registers/addresses locations. This does not affect their "enable" status.

Peripheral names can be used in the place of register names to enable or display all the registers associated with that peripheral. The valid peripheral names for the selected device can be obtained by using the Simulator's **help periph** command. The word `all` can be used to enable or display all of the device registers.

`V``(version)`

Entering the **display** command with the single parameter `V` will display the Simulator version number in the Session window.

Table 12-13. DISPLAY Commands

Command	Resulting Simulator Action
`display on`	Enables all registers for display
`display on pi:0..20 xi:30..40`	Enable internal p memory address block 0 to 20 and internal x memory address block 30 to 40 for display.
`display off` `display on core ssi0`	Disables display of all registers and addresses, then enables display of the DSP core registers and the SSI0 peripheral registers.
`display w r0..r3 x:0..100`	Displays registers **r0** through **r3** and x memory locations 0 through 100 if they have been written to since the last display.

Table 12-14. DISPLAY Commands Without Enable Keywords

Command	Resulting Simulator Action
`display`	Immediately displays all currently enabled registers and memory.
`display p:0..300`	Immediately displays p memory addresses 0 through 300.
`display test.asm@7`	Immediately displays memory location corresponding to line 7 of source file test.asm.
`display xdat`	Immediately displays memory location corresponding to symbolic label xdat.
`display all`	Immediately displays all registers plus the enabled memory locations.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.11 down - Move Down the C Function Call Stack

DOWN [n]

The **down** command is used to move down the call stack. It can be used in conjunction with the **where**, **frame**, and **up** commands to display and traverse the C function call stack.

After entering a new call stack frame using down, that call stack frame becomes the current scope for evaluation. This means that for C expressions, the **evaluate** command acts as though this new frame is the proper place to start looking for variables.

[n] Optional. The number of frames to move down. If no number is specified, the Simulator moves down one frame in the call stack.

Table 12-15. DOWN Commands

Command	Resulting Simulator Action
down	Moves down the call stack by one stack frame.
down 2	Moves down the call stack by two stack frames.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.12 evaluate - Evaluate an Expression

EVALUATE [**B**(binary)/**D**(dec)/**F**(float)/**H**(hex)/**U**(unsigned)] expression/**{**c_expression**}**

The **evaluate** command is used as a calculator for evaluating arithmetic expressions or for converting values from one radix to another.

[**B**(binary)/**D**(dec)/**F**(float)/**H**(hex)/**U**(unsigned)]

Optional. Specifies the radix in which the result of the expression will be displayed. If a radix is not specified in the **evaluate** command line, the current default radix (specified by the **radix** command) will be used.

expression/**{**c_expression**}**

An expression consists of an arithmetic combination of operators and operands. An operand can be a register name, a memory location, or a constant value.

The order of evaluation of an expression's operators will be associated from left to right. Parentheses can be used to force the order of evaluation of the expression. A more extensive discussion of the expressions which are valid for the **evaluate** command is presented in Chapter 5, "Device I/O and Peripheral Simulation," on page 5-1.

When values held in the device's registers or memory spaces are used in an expression that involves a multiply operator, the display radix (specified by the **radix** command) will determine whether the operation executed is a floating point or integer multiply.

Table 12-16. EVALUATE Commands

Command	Resulting Simulator Action
`evaluate r0+p:$50`	Adds the value in r0 register to the value in program memory address hexadecimal 50 and displays the result using the default radix.
`evaluate b $345`	Converts hexadecimal 345 to binary and displays the result.
`evaluate lab_d`	Displays the address of the location associated with symbolic label **lab_d**.
`evaluate {count}`	Displays the value of the C variable **count**.
`evaluate h %10101010&p:r0`	Calculates the bitwise **AND** of the program memory address specified by the value in r0 register and the binary value 10101010 and displays the result in hexadecimal.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.13 finish - Execute Until End of Current Subroutine

FINISH

The **finish** command executes instructions until the current subroutine or function call is completed. This is useful if execution has for some reason been halted in the middle of a subroutine or function.

12.14 frame - Select C Function Call Stack Frame

FRAME [#n]

The **frame** command is used to select the current call stack frame. It can be used in conjunction with the **where**, **down**, and **up** commands to display and traverse the C function call stack.

[#n] Optional. Specifies the number of the frame in the call stack which will be selected as the current call stack frame.

After entering a new call stack frame using the **frame** command, that call stack frame becomes the current scope for evaluation.

The **frame** command executes instructions until a return-from-subroutine (RTS) instruction is executed within the current subroutine. The Simulator simply steps, checking if any instruction is an RTS. If so, that RTS is executed, and instruction execution halts immediately afterward. While stepping, if a branch to subroutine or jump to subroutine instruction is encountered, tests for the RTS instruction are suspended until execution resumes at the address following the subroutine call..

Table 12-17. FRAME Commands

Command	Resulting Simulator Action
frame #2	Selects call stack frame number two.
frame #0	Selects call stack frame number zero (innermost frame).

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.15 go - Execute DSP Program

GO [(from)location/**R**(reset)] [(to break number)#bn] [(occurrence):count]

The **go** command begins execution of DSP code. The Simulator fetches, de-codes, and executes instructions in the exact manner as the device that is being simulated. The **go** command will pass control to the Simulator until a breakpoint is reached, or until program execution is otherwise stopped.

Entering the **go** command with no parameters will start simulation from the current program counter value.

[(from)location/**R**(reset)]

Optional. The location parameter represents the address, symbolic label or line number (if symbolic debug information has been loaded) from which execution will begin.The reset (**R**) parameter causes a simulation of the reset sequence in the processor. The device registers are reset and execution begins at the reset exception address. If an address or reset parameter is included, the instruction pipeline, instruction counter, and cycle counter will be cleared before program execution begins.

[(to break number)#bn]

Optional. The optional #bn parameter may be used to cause execution to halt if the breakpoint condition represented by break number #bn occurs. All other breakpoint conditions are ignored.

[(occurrence):count] Optional. The :count parameter causes execution to halt only if the breakpoint has occurred count times. If #bn is not specified, simulation will stop if count number of breakpoint conditions have occurred.

Table 12-18. GO Commands

Command	Resulting Simulator Action
go	Starts program simulation from the current instruction. Stops at the first occurrence of any breakpoint.
go $100	Starts program simulation at program memory address hex 100 after clearing the instruction pipeline. Stops at the first occurrence of any breakpoint.
go r	Clears the Simulator pipeline and starts program execution at the reset vector. The simulated machine state is also reset according to the processor reset sequence.
go #5	Begins execution from the current instruction. Halts on the first occurrence of breakpoint number 5.
go #5 :3	Begins execution from the current instruction. Halts on the third occurrence of breakpoint number 5.
go lab_d #5 :3	Starts execution from symbolic address **lab_d** after clearing the Simulator pipeline. Halts on the third occurrence of breakpoint number 5.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.16 help - Simulator Help Text

HELP [command/reg/topic]

The **help** command provides syntax and examples of Simulator commands, descriptions of device register bit fields, and help on other topics related to device or Simulator operation. If no keyword is entered the Simulator displays a summary of the possible help topics.

[command/reg/topic] Optional. The command, register, or topic for which to display help. If the keyword is a command name the Simulator displays a summary of that command's parameters along with a brief description and examples. If the keyword is a register name the Simulator displays the specified register's contents along with the help text associated with the register.

Table 12-19. HELP Syntax

Command	Resulting Simulator Action
help	Displays a summary of all available commands and their parameters.
help asm	Displays a summary of the assemble command and its parameters.
help omr	Displays the contents of the DSP's Operating Mode Register.

The topic keywords in Table 12-20 provide online help for the described topics.

Table 12-20. List of Help Topic Keywords

Keyword	Help Topic
io	list of on-chip io registers and their addresses
int	list of interrupt vector addresses for the device
periph	list of peripheral names
pin	list of pin names and numbers, and the current pin states
port	list of port names
mode	initial chip operating mode summary
map	memory map descriptions for various omr settings
mem	memory names with block addresses
sym	display program symbol table names and values
reg	display register size, register and peripheral index
stack	display of values on the device stack

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.17 history -Disassemble Previously Executed Instruction

HISTORY

The **history** command disassembles and displays the previous 32 instructions executed by the device. The instructions are displayed in the order that they were executed, with the most recent instruction appearing at the bottom of the list. The last instruction in the list has been fetched and decoded by the device and will enter the execute phase in the next device cycle. It is in the same state as instructions that are disassembled and displayed at the end of each trace display.

A typical use for this command would be to determine the sequence of instructions that terminated in a user-defined breakpoint. You would set the breakpoint condition using the **break** command, then issue the **go** command. When the break condition is met, instruction execution halts and the currently enabled registers are displayed. You can then issue the **history** command to view the last 32 instructions that executed prior to the breakpoint.

The device execution history can also be logged continuously to an output file using the Simulator **output** command.

12.18 input - Assign Input File

INPUT [#n] [**T**(timed)] addr/port/periph/pin[_group] **OFF/TERM**/file [**-RD/-RF/-RH/-RU**]

INPUT [#n] pin (from)[**DV**n:]pin

INPUT [#n] addr (from)[**DV**n:]addr

The **input** command causes the retrieval of data from a source that you specify to a peripheral, memory location, or device pin . The input data is in ASCII format. Valid port, peripheral and pin names for the device can be listed with the **help port**, **help periph** and **help pin** commands.

There are three forms of the **input** command. The first form allows input to a memory location, port, peripheral, or pin from data provided either in a file or from the keyboard. The second form allows input to a device pin from another device pin without having to store the data in a disk file. The third form of the **input** command causes the Simulator to read the memory location of a specified source (specified by (from)[**dv**n:]addr) each time the destination memory address is accessed for a read. This enables simulation of a system consisting of multiple devices via dual-port memory.

Data values may be expressed in hexadecimal, decimal, or floating point for memory and peripheral name assignments, as one of 5 input values (0,1,n,p or X) for assignment to individual digital pins, or as single precision floating-point values for analog input pins.

[#n]	Optional. The number that will be assigned to this input. Input numbers do not have to be consecutive - they can be assigned arbitrarily, which means that you can assign input numbers in any way that helps you organize the sources of the inputs. If no number is provided, the Simulator will automatically assign the next available number.
[**T**(timed)]	Optional. When included with the **input** command, this parameter indicates that the source data is in the form of time-data pairs.

addr/port/periph/pin[_group]

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

The destination of the input data. All subsequent reads of the destination will reference the source of the input data.

OFF/TERM/file — The source of the input data. Providing the **OFF** parameter disables the input specified by #n. Providing the **TERM** parameter indicates that the input data will be typed from the keyboard. Providing a filename indicates that the input data is contained in file. If a filename suffix is not specified, the Simulator will assume ".io" for a non-timed input file and ".tio" for a timed input file.

[**-RD/-RF/-RH/-RU**] — Optional. The default input radix for the data may be specified by using **-RD** for decimal, **-RF** for fractional, **-RH** for hexadecimal, or **-RU** for unsigned. Hexadecimal input is the default for addr, port and peripheral data values. Input analog pin data files must be assigned with the **-RF** radix designator, and the file data must be single precision floating point values. The time value is always expressed in decimal. Chapter 3, "Object Files and Data Files," on page 3-1, contains an extensive description of the input file format.

pin — In the second form of the **input** command, this parameter represents the name of the pin that is the destination of the input data. The second form of the **input** command allows input to a device pin from another device pin without having to store the data in a disk file.

(from)[**DV**n:]pin — In the second form of the input command, this parameter represents the name of the pin that is the source of the input data. The source pin may optionally be preceded by a device number to allow the simulation of pin to pin connections in systems consisting of multiple devices.

addr — Used with the third form of the **input** command, this parameter represents the destination address. All subsequent reads of this memory address will reference the address specified as the input source.

(from)[**DV**n:]addr — Used with the third form of the **input** command, this parameter represents the address of the source of the input data. The address may optionally be preceded by a device number (**DV**n). The third form of the command causes the Simulator to read the memory location of a specified source (specified by **dv**n:addr) each time the destination memory address is accessed for a read. This enables simulation of a system consisting of multiple devices via dual-port memory. The source device must exist (new devices can be added with the **device** command) prior to issuing this form of the **input** command.

Table 12-21. INPUT Commands

Command	Resulting Simulator Action
input xe:$800 xfile -rd	Gets values for external memory location x:800 from input file "xfile.io". The data values are stored in decimal form in the input file.
input ssi0 hfile	Gets values for the SSI0 peripheral from input file "hfile.io".

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Command	Resulting Simulator Action
`input d15..d0 dfile`	Gets values for pins D15 through D0 from input file "dfile.io".
`input d15..d0 dfile`	Gets values for pins D15 through D0 from input file "dfile.io".
`input irqb dv1:pb0`	Inputs values for the current device's irqb pin from device dv1's pb0 pin.
`input t irqa term`	Inputs time and data pairs from the terminal for the device IRQA pin.
`input x:500 dv5:x:3000`	Inputs data for memory reads of x:500 of the current device from device number 5 address x:3000.
`input #2 x:$800 xfile -rd`	Gets values for external memory location x:800 from input file "xfile.io". The data values are stored in decimal form in the input file. Input assignment number 2 is explicitly replaced due to the #2 in this command form.
`input #2 off`	Inputs assignment number 2 is explicitly deleted by index number.
`input mic micfile -rf`	Inputs untimed analog pin data for the mic analog pin from the file "micfile.io".

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.19 list - List Source File Lines

LIST [+/-/./addr]

The **list** command displays source lines or disassembled instructions from the specified source file, or beginning at the specified address.

The current display mode determines whether a source file or assembly mnemonics will be displayed. If the Simulator is in the register display mode, this command will switch it to the source display mode and display the source file lines associated with the specified address or line number. If the display mode is already source or assembly, the display mode is not altered. The assembly display mode displays disassembled instructions corresponding to the specified address or line number.

[+/-/./addr] Optional. The next or previous pages of the currently displayed source file may be selected by specifying "+" or "-" . In addition, the source or assembly associated with the current execution address may be selected by specifying "." (period) or by using the **list** command without a parameter.

If the addr parameter is used with the **list** command, the disassembled instructions from the specified address will be displayed. Alternatively, a line number can be provided in place of addr (symbolic debug information must be loaded).

Table 12-22. LIST Commands

Command	Resulting Simulator Action
list 20	Lists source or assembly corresponding to line 20 of the current source file.
list test.asm@20	Lists source or assembly corresponding to line 20 of the source file test.asm.
list test.asm	Lists source or assembly corresponding to line 1 of the source file test.asm.
list +	Displays the next page of the current source file or assembly.
list .	Displays source or assembly corresponding to the current execution address.
list-	Displays the previous page of the current source file or assembly.
list test.asm	Lists source or assembly corresponding to line 1 of the source file test.asm.
list lab_1	Lists source or assembly corresponding to symbolic address lab_1.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.20 load - Load DSP Files or Configuration

LOAD [S(state)|M(memory-only)|D(debug symbols-only)] (from)file

The **load** command can be used to load DSP object module format (.lod) files or DSP COFF (.cld) files into the Simulator memory or to load a previously saved simulation state file.

[S(state)|M(memory-only)|D(debug symbols-only)]

Optional. If the S key character is specified, the Simulator will load filename as a Simulator state file. The Simulator state file can be created using the Simulator **save s** command. Loading the Simulator state changes the entire setup of the Simulator to the previous definition saved in the state file.

If the M key character is specified, the Simulator will load object file without modifying the Simulator' s symbolic debug information.

If the D keycharacter is specified, the Simulator will load only the symbolic debug information from the object file. The device memory contents are not altered. Only the COFF format files (.cld suffix) are supported by this option.

(from)file

If only a file parameter is specified, the Simulator assumes that the file is an object file. The object file may be in either the special ASCII OMF format Chapter 3, "Object Files and Data Files," on page 3-1, or in the DSP COFF format generated by the DSP Assembler. The OMF format file can be created using the Simulator **save** command or with a text editor. A directory path may be specified with the filename. If no filename suffix is specified, the Simulator will search first for an OMF format ".lod" file, then for a COFF format ".cld" file. Loading a COFF format file replaces the Simulator's symbolic debug information unless the M option is specified.

If the S parameter is specified but no filename suffix is specified, the ".sim" file extension is assumed.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Table 12-23. LOAD Commands

Command	Resulting Simulator Action
`load \source\testloop.obj`	Loads the OMF format "testloop.obj" file from directory "source".
`load \source\testloop.cld`	Loads the COFF format "testloop.cld" file from directory "source", including the memory contents and any symbolic debug information contained in the file.
`load lasttest`	Loads the OMF format "lasttest.lod" file from current directory.
`load d test.cld`	Loads the symbolic debug information from the COFF format "test.cld" file, ignoring the memory contents of the file.
`load m test.cld`	Loads the COFF format "test.cld" file, ignoring any symbolic debug information in it.
`load s lunchbrk`	Loads "lunchbrk.sim", replacing the entire current Simulator state.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.21 log - Log Commands, Session, Profile

LOG [**OFF**] [**C**(commands)/**S**(session)/**P**(profile) [file [-**A**/-**O**/-**C**]]]

LOG [**OFF**] **V**(source display status line)

The **log** command allows you to record command entries, to record all session display output, or to record an analysis of DSP program execution. Record of commands is useful as a method of generating macro command files. Recording all session display output provides a convenient way for you to review the results of an extended sequence of commands. Since the output log files are in ASCII format, they can easily be printed or reviewed using an editor program. Recording a program profile assists in the analysis of program structure and execution characteristics.

Entering the **log** command with no parameters will cause the Simulator to display the currently opened log filenames.

[OFF] Optional. The OFF parameter is used to terminate logging.

[**C**(commands)/**S**(session)/**P**(profile) [file [-**A**/-**O**/-**C**]]]

The C and S parameters are used to specify whether the logfile will contain only commands (C), or all session output (S). A keycharacter **-A, -O,** or **-C** may be specified to select append, overwrite, or cancel if the filename already exists. If no append/overwrite option is included with the **log** command, you will be prompted during command execution. If no suffix is specified for file, the suffixes ".cmd" is automatically added to the command log file; similarly ".log" is automatically added to the session log file.

The P parameter specifies that a program execution profile is to be created. The program to be profiled must be loaded before issuing the 'log p' command. Both memory and symbols must be loaded. Information is gathered as program execution is simulated, and the profile output files are written when the profiling is terminated with the command **log off p**. For profile logging, two suffixes are used, a ".log" file which contains plain text which may be printed on any 80-column printer, and a ".ps" file which is formatted for a postscript printer.

V(source display status line)

Used with the second form of the **log** command, the V parameter enables logging of the source display status line to a session log file. It is primarily intended for testing the Simulator display.

In multiple device simulations, there is a separate session log file associated with each simulated device, but there is only a single command log associated with the entire multiple device simulation. If a device type is changed using the **device** command, the user interface information associated with the discarded device type, including the session log file name, is cleared; so it is best to specify the **log s** command following the **device** command for a particular device.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 12-24. LOG Commands

Command	Resulting Simulator Action
`log`	Displays currently opened log files.
`log s \debugger\session1`	Logs all display entries to session1.log in directory \debugger
`log c macro1 -a`	Logs all commands to the file "macro1.cmd". Append if it already exists.
`log off c`	Terminates command logging.
`log off`	Terminates all logging.
`log v` `log s session1`	Logs source display status line and all display entries to "session1.log"
`load px41v17.cld` `log p px41v17 -o`	Loads memory and symbols for program px41v17 and log the program profile in files px41v17.log and px41v17.ps. Overwrites these files without warning if they already exist. Note that the program(s) to be profiled must have been loaded before this **log command** is issued.

12.22 more - Enable/Disable Session Paging Control

`MORE [OFF]`

The **more** command allows you to enable or disable the paging of data in the Session window. This is particularly useful when displaying large amounts of data and you wish to examine page by page.

`[OFF]` Optional. Turns off paging.

The paging feature is turned off by default. Data will scroll vertically across the screen when it is larger than the size of the screen.

Table 12-25. MORE Commands

Command	Resulting Simulator Action
`more`	Turns on session display paging control.
`more off`	Disables session display paging control (reset or default state).

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.23 next- Step Over Subroutine Calls or Macros

NEXT [count] [**LI**(lines)/**IN**(instructions)] [**H**(halt at breakpoints)]

The **next** command functions the same as the **step** command, except that if the next instruction to be executed calls a subroutine or begins execution of a macro, all the instructions of the subroutine or macro are executed before stopping to display the enabled registers. In order to recognize macros, the symbolic debug information for the program code must be loaded.

[count] Optional. When included, the count parameter indicates the number of lines/instructions to execute before halting execution. If no count is specified, the default count is 1.

[**LI**(lines)/**IN**(instructions)]

Optional. As the default, the **next** command executes the next instruction if viewing the assembly or register screens, and the next line if viewing the source screen. The **li** and **in** parameters permit source line or instruction increments to be specified explicitly.

[**H**(halt at breakpoints)]

Optional. As the default, all breakpoints are ignored while the **next** command is executing. The h parameter enables halting at breakpoints.

Table 12-26. NEXT Commands

Command	Resulting Simulator Action
next	Steps over subroutine calls or macros; or otherwise just advances one instruction or source line, depending on the display mode, and displays the enabled registers and memory blocks.
next li	Steps over subroutine calls or macros; or otherwise just advances one source line and displays the enabled registers and memory blocks.
next in	Steps over subroutine calls or macros; or otherwise just advances one assembly instruction and displays the enabled registers and memory blocks.
next 10	Executes the equivalent of the next 10 instructions, halting to display the enabled registers and memory blocks only after the tenth instruction is completed.
next 10 li	Executes the equivalent of the next 10 lines, halting to display the enabled registers and memory blocks only after the tenth line is completed.
next 10 h	Executes the equivalent of the next 10 instructions, halting to display the enabled registers and memory blocks after the tenth instruction is complete, or when a breakpoint is encountered.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.24 output - Assign Output File

OUTPUT [#n] [**T**] addr/port/periph/pin[_group] **OFF**/**TERM**/file [-**RD**/**-RF**/**-RH**/**-RU**/**-RS**] [**-A**/**-O**/**-C**]

OUTPUT [#n] [**T**] **history OFF**/**TERM**/file [**-A**/**-O**/**-C**]

OUTPUT [#n] [**T**] **ehistory OFF**/**TERM**/file [**-A**/**-O**/**-C**]

The **output** command stores data from a peripheral, memory location, or device pin to a specified destination. The output data is in ASCII format. Valid port, peripheral and pin names for the device can be listed with the **help port**, **help periph** and **help pin** commands.

There are three forms of the output command. The first form allows output from a memory location, port, peripheral, or pin to either a `file` or to the Session window (**TERM**). The second form of the **output** command creates a continuous log of the device execution addresses and disassembled opcodes. The output format is similar to the output generated by the Simulator **history** command.

The third form of the command, which specifies **ehistory**, is an extended version of the **output history** command. Additional execution history information is logged to the output file. Only one of output history or output ehistory may be active.

[#n] — Optional. The number that will be assigned to this output. Output numbers do not have ot be consecutive - they can be assigned arbitrarily, which means that you can assign output numbers in any way that helps you organize the outputs. If no number is provided, the Simulator will automatically assign the next available number.

[**T**] — Optional. When included with the **output** command, this parameter indicates that the output data is in the form of time-data pairs.

addr/port/periph/pin[_group] — The source of the data. Assignment to a memory address causes all subsequent writes of that memory address to store data in the output file/Session window.

OFF/**TERM**/file — The destination of the output data. Providing the **OFF** parameter disables the output specified by #n. Providing the **TERM** parameter indicates that the output data will be typed to the Session window. Providing a filename indicates that the output data will be written to `file`. If a filename suffix is not specified, the Simulator will assume "`.io`" for a non-timed output file and "`.tio`" for a timed output file.

[-**RD**/**-RF**/**-RH**/**-RU**/**-RS**] — Optional. The default output radix for the data may be specified by using **-RD** for decimal, **-RF** for fractional, **-RH** for hexadecimal, **-RU** for unsigned, or **-RS** for character string. The **-RF** radix, when specified for a single output pin, will output the analog single precision floating point value associated with the pins analog function. The **-RS** radix is valid only for output memory locations. It interprets values written to the specified memory location to be the address of a null terminated character string in the same memory space.The character string will be displayed or

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

written to an output file. This string radix is provided primarily for use when debugging programs created with the C Compiler. The output time value is always expressed in decimal. Chapter 3 contains a thorough description of the output file format.

[-A/-O/-C] — Optional. The **-A, -O**, or **-C** parameter may be used to indicate append, overwrite, or cancel if the filename already exists. If you do not specify this parameter, and the filename already exists, you will be prompted during command execution to make the decision to append, overwrite, or cancel.

history — Used with the second form of the **output** command. This usage creates a continuous log of the device execution addresses and disassembled opcodes. The output format is similar to the output generated by the Simulator **history** command.

ehistory — Used with the third form of the **output** command. This usage is an extended version of the **output history** command. Additional execution history is included in the output, including device wait state cycles and bus arbitration cycles and indication of other stall conditions. The extra information is preceded by double asterisks in the log file.

Table 12-27. OUTPUT Commands

Command	Resulting Simulator Action
output x:$0 xfile -rd	Stores values written to memory location x:0 in output file "xfile.io". The data values will be stored in decimal.
output ssi0 ssi0file -a	Stores values from the SSI0 peripheral to output file "ssi0file.io". Appends to the file if it already exists.
output a15..a0 afile	Stores output values for address pins a15 through a0 to output file "afile.io".
output #2 t bg term	Outputs time and data pairs from the device BG pin to the terminal. Output number 2 is explicitly replaced by this command.
output #2 off	Output number 2 is explicitly turned off by index number reference.
output spkp spfile -rf	Outputs untimed single precision floating point values for the analog pin spkp to the output file "spfile.io".
output xdat1 xfile	Stores values written to memory location associated with the symbolic label xdat1 to the output file "xfile.io". The data values will be stored in the default hexadecimal radix.
output history hisfile	Stores device execution history to output file "hisfile.io".
output ehistory hisfile	Stores extended device execution history to output file "hisfile.io".

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.25 path - Specify Default Pathname

PATH [pathname]

PATH +pathname[,pathname,...]

PATH -

The **path** command defines the default pathname that is used by the Simulator to store temporary files, log files, macro command files, object files, and peripheral I/O files. If no parameters are specified, the current default pathname is displayed in the Session window. The default pathname can be overridden by explicitly specifying a pathname as a prefix to the filename in any of the commands which reference a file.

[pathname] Optional. Used in the first form of the path command, indicates the directory to use as the pathname.

+pathname[,pathname,...]

Used in the second form of the **path** command. Alternate source pathnames may be specified using the **path +** form of the command. Each time the command is issued, the specified pathname, or comma-separated list of pathnames, is added the current list. When searching for files, the Simulator will first search the default pathname, then search in each of the alternate source pathnames, in the order that they were specified.

- The third form of the command, "path -", deletes the entire list of alternate source path-names.

Table 12-28. PATH Commands

Command	Resulting Simulator Action
path \sim	Defines the default working directory for Simulator files as "\sim".
path \sim\day2	Defines the default working directory for Simulator files as "\sim\day2".
path + ..\src	Adds pathname "..\src" to the list of alternate source pathnames.
path + ..\src,..\src2	Adds pathnames "..\src" and "..\src2" to the list of alternate source pathnames.
path –	Clears the list of alternate source pathnames.
path	Shows the default working directory and help file directory for the current device, and the list of alternate source pathnames.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.26 quit - Quit Simulator Session

`QUIT [E(enable)/D(disable)]`

The **quit** command passes control back to the operating system after closing all log files, input and output files, and macro files.

`[E(enable)/D(disable)]` Optional. Quit enable (**E**) and quit disable (**D**) control the action taken by the Simulator if an error occurs during the execution of a macro command. The **quit enable** specifies that the macro command should be aborted and the Simulator quit immediately with a non-zero exit status. The **quit disable** command specifies that the Simulator should not exit.

.

Table 12-29. QUIT Commands

Command	Resulting Simulator Action
`quit`	Closes all currently open files and returns to the Operating System.
`quit e`	Specifies that errors in a macro command will cause the Simulator to exit with a non-zero status.

12.27 radix - Change Input or Display Radix

`RADIX [B(binary)/D(dec)/F(float)/H(hex)/U(unsigned)] [reg[_block]/addr[_block]]...`

The **radix** command allows you to change the default number base for command entry or for display of registers and memory locations. The Simulator, by default, uses decimal input radix and hexadecimal display radix when it is initially invoked. Changing the default input radix allows you to enter constants in the chosen radix without typing a radix specifier before each constant.

If no parameters are used with the radix command, the current default radix is displayed in the Session window.

`[B(binary)/D(dec)/F(float)/H(hex)/U(unsigned)]`

Optional. This parameter, when not followed by a register name or memory location, specifies the radix to use as the default input radix. When followed by a register name or memory location, the radix will be applied as the default display radix when displaying the values conatined in the register/memory location.

`[reg[_block]/addr[_block]]...`

Optional. Specifying a list of register and/or memory locations following the radix specifier will set the display radix of the registers and memory to the radix specified. This does not affect the default input radix.

A range of registers can be specified by typing `start_reg..end_reg`, where `start_reg` is the beginning register and `end_reg` is the last register of the block.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

A range of addresses can be specified by typing `start_addr#count`, where `start_addr` is the first address of the block and `count` represents the size of the block. An address block can also be specified by typing `start_addr..end_addr`. For example: `p:5#20` indicates 20 locations beginning from program memory location 5; `p:5..24` indicates program memory locations 5 through 24.

The radix command is used to define the default number base. However, the default radix can be overridden on a individual usage basis. Hexadecimal constants can always be specified by preceding the constant with a dollar sign ($). Likewise, a decimal value can be specified by preceding the constant with a grave accent (‘), and a binary value may be specified by preceding the constant with a percent sign (%).

Table 12-30. RADIX Commands

Command	Resulting Simulator Action
`radix`	Displays the default input radix currently enabled.
`radix h`	Changes default input radix to hexadecimal. Hexadecimal constant entries no longer require a preceding dollar sign, but any decimal constants will require a preceding grave accent.
`radix f x:0..10 x0 y0 a b`	Change the display radix for the specified registers and memory blocks to floating point.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.28 redirect - Redirect stdin/stdout/stderr for C Programs

REDIRECT **STDIN OFF**/file

REDIRECT **STDOUT/STDERR OFF**/file [**-A/-O/-C**]

REDIRECT [**OFF**]

The **redirect** command is used to redirect the stdin/stdout/stderr for C programs. It allows you to redirect stdin from a file, and redirect stdout/stderr to files. Performing the **redirect** command with no parameters will display the status of each I/O stream in the Session window.

Beware that no I/O processing or handling of redirection occurs if I/O streaming has been disabled with the **streams** command.

STDIN — Used with the first form of the **redirect** command. The **STDIN** parameter indicates that the "standard input" is to be redirected.

OFF/file — Used with the first and second forms of the **redirect** command. The **OFF** parameter indicates that the redirection of the specified input/output is to be turned off. In the context of the first form of the **redirect** command, the file parameter indicates the name of the file from which "standard input" will be provided. In the context of the second form of the **redirect** command, the file parameter indicates the name of the file to which "standard output" or "standard error" will be written.

In all cases, if no file extension is specified when specifying file with the **redirect** command, the Simulator will assume the ".cio" file extension.

STDOUT/STDERR — Used with the second form of the **redirect** command. The **STDOUT/STDERR** parameter indicates that "standard out" or "standard error" is to be redirected.

[**-A/-O/-C**] — Optional. The **-A, -O**, or **-C** parameter may be used to indicate append, overwrite, or cancel if the filename file already exists. If you do not specify this parameter, and the filename already exists, you will be prompted during command execution to make the decision to append, overwrite, or cancel.

[**OFF**] — Optional. Used with the third form of the redirect command. When used as the only parameter, the **OFF** parameter turns off all redirected **STDIN**, **STDOUT**, and **STDERR**.

Table 12-31. REDIRECT Commands

Command	Resulting Simulator Action
redirect	Displays the redirect list, which shows each of the three streams that can be redirected, along with where they are being redirected to.
redirect stdin input	Redirects the C stdin (standard input) stream from the file input.cio (.cio is the default ex-tension).

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Command	Resulting Simulator Action
`redirect stdout output.txt`	Redirects the C stdout (standard output) stream to the file output.txt.
`redirect stderr errors`	Redirects the C stderr (standard error) stream to the file errors.cio.
`redirect stdout output -o`	Redirects the C stdout stream to the file output.cio, overwriting the file if it already exists.
`redirect stdout output -a`	Redirects the C stdout stream to the file output.cio, appending to the end of the file if it already exists.
`redirect stdout output -c`	Redirects the C stdout stream to the file output.cio, but doesn't redirect if the file already exists.

12.29 reset - Reset Device or State

RESET **S**(state)/**D**(device) [mode]

The **reset** command can be used to reset the device registers (D) or the entire Simulator state (S). It can also be used to select the operating mode that the device will be set to in response to a simulated hardware reset sequence.

S(state)/**D**(device) — The **S** parameter indicates that the the entire Simulator state should be reset to the start-up condition. All breakpoints are cleared, the memory is initialized, and all logging and I/O files are closed.

The **D** parameter indicates that the device should be reset to the start-up condition. All device registers are reset to the defined reset conditions.

[mode] — Optional. The mode parameter specifies the DSP operating mode in the form **M**n (n=decimal digit).

To see the Operating Modes that are available for a particular device:

1. At the command line, type:

   ```
   help mode
   ```

2. View the Session window. It will contain a list of operating modes available to the particular device that you are simulating.

Table 12-32. RESET Commands

Command	Resulting Simulator Action
`reset d`	Resets all device registers to the defined reset conditions.
`reset d m0`	Resets the device registers and select operating mode 0 as the default operating mode following subsequent hardware reset sequences.
`reset s`	Resets the entire Simulator state to the start-up condition. All breakpoints are cleared, the memory is initialized, and all logging and I/O files are closed.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.30 save - Save Simulator File

SAVE **S**(state)/addr[_block]... filename [**-A**/**-O**/**-C**]

The **save** command allows creation of a Simulator state file from the current Simulator state, or creation of an object module format file or a COFF format file from specified memory blocks.

S(state)/addr[_block]...

If S is specified as the second parameter, a Simulator state file is created. It contains the entire simulation state, including memory contents, breakpoint settings, and the current pointer position of any open files. This file is in an internal format that is efficient for the Simulator to store and load (see the load s command description). The default suffix for a Simulator state filename is ".sim".

If the addr[_block] parameter is used, the memory areas specified are stored in object format so the file can be reloaded with the Simulator **load** command. The default object format is the OMF format described in Chapter 6. If no file extension is explicitly specified, the extension for an OMF file ".lod", will be appended to filename . If the COFF file extension, ".cld", is specified explicitly in the filename, the memory contents will be stored in the DSP COFF object file format. The Simulator does not store symbolic debug information in the output COFF object file.

An address block can be specified by typing start_addr#count, where start_addr is the first address of the block and count represents the size of the block. An address block can also be specified by typing start_addr..end_addr. For example: p:5#20 indicates 20 locations beginning from program memory location 5; p:5..24 indicates program memory locations 5 through 24.

filename — The name of the file to be saved.

[**-A**/**-O**/**-C**] — Optional. The **-A**, **-O**, or **-C** parameter may be used to indicate append, overwrite, or cancel if the file filename already exists. If you do not specify this parameter, and the filename already exists, you will be prompted during command execution to make the decision to append, overwrite, or cancel. Appending is not a valid option for state files.

Table 12-33. SAVE Commands

Command	Resulting Simulator Action
save p:0..$ff x:0..$20 session1 -a	Saves all three memory maps to OMF file "session1.lod". If the file already exists, append to it.
save s lunchbrk	Saves the Simulator state to filename "lunchbrk.sim".
save s lunchbrk.b -c	Saves the Simulator state to filename "lunchbrk.b". If the file already exists, cancel this command.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.31 step - Step Through DSP Program

STEP [count] [**CY**(cycles)/**LI**(lines)/**IN**(instructions)] [**H**(halt at breakpoints)]

The **step** command allows you to execute count instructions or clock cycles before displaying the enabled registers and memory blocks. This command gives you a quick way to specify execution of a number of instructions without having to set a breakpoint. It is similar to the **trace** command except that display occurs only after the count number of cycles or instructions have occurred.

Unlike the **next** command, if the next instruction to be executed calls a subroutine or begins execution of a macro, the instructions of the subroutine or macro are counted towards the total number of instructions that were specified to be executed with the **step** command. If execution stops in the middle of a subroutine or function, execution can continue to the end of the subroutine or function by using the **finish** command.

[count] Optional. When included, the count parameter indicates the number of cycles/lines/instructions to execute before halting execution. If no count is specified, the default count is 1.

[**CY**(cycles)/**LI**(lines)/**IN**(instructions)]

Optional. As the default, the **step** command steps in instruction increments if viewing the assembly or register screens, and in source line increments if viewing the source screen. The **CY**, **LI** and **IN** options permit cycles, source line or instruction increments to be specified explicitly.

[**H**(halt at breakpoints)] Optional. The default is to ignore all breakpoints while the **step** command is executing. The h parameter enables halting at breakpoints.

Table 12-34. STEP Commands

Command	Resulting Simulator Action
step	Steps one instruction or source line, depending on the display mode, and displays the enabled registers and memory blocks.
step li	Steps one source line, regardless of the display mode, and displays the enabled registers and memory blocks.
step $50	Executes hex 50 instructions or source lines, depending on the display mode, then stops and displays the enabled registers and memory blocks at the end of the hex 50th instruction.
step $50 in h	Executes hex 50 instructions, regardless of the display mode, then stops and displays the enabled registers and memory blocks at the end of the hex 50th instruction. Halts if a breakpoint is encountered during the execution.
step 20 cy	Executes 20 clock cycles and displays the enabled registers and memory blocks at the end of the 20th clock cycle.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.32 streams - Enable/Disable Handling of I/O for C Programs

STREAMS [**E**NABLE/**D**ISABLE]

The **streams** command is used to enable and disable the handling of input and output on the host side for C programs. By default, streaming is enabled. When enabled, all input and output that is done in the C program running on the device is handled on the host side. For example, when an fopen() call is made in the C program running on the DSP call, the host software intercepts the call and does the fopen() on the host side.

If the streams command is performed with no parameters, the current status of streaming is displayed in the Session window.

[**E**NABLE/**D**ISABLE] Optional. Enables or disables streaming..

Table 12-35. STREAMS Commands

Command	Resulting Simulator Action
streams e	Enables handling of C input/output. All input/output calls done in a C program running on the DSP will be handled by the host software (e.g. fopen(), fwrite(), printf(), etc.).
streams d	Disables handling of C input/output.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.33 system - Execute System Command

SYSTEM [**-C**(continue immediately)] [system_command [parameter_list]]

The **system** command allows you to execute an operating system command from within the Simulator. Operating system commands invoked from within the Simulator are not logged to the Session window for review.

[**-C**(continue immediately)]

Optional. The **-C** parameter causes control to return to the Simulator after the operating system command is completed without prompting you. This may be useful in macro commands, allowing system commands to be used without requiring operator intervention. If the **-C** parameter is not specified, before control returns to the Simulator you are prompted to `Hit return to continue.` This allows you to inspect the command output before it is destroyed.

[system_command [parameter_list]]

Optional. If a `system_command` is included in the **system** command, the specified `system_command` is executed, including any parameters specified in the `parameter_list`. If the command line does not contain a `system_command`, then a mode is entered in which multiple **system** commands may be entered. Return to the Simulator occurs when you enter **exit** on the operating system command line.

Table 12-36. SYSTEM Commands

Command	Resulting Simulator Action
`system dir`	Executes the system "dir" command and immediately returns to the Simulator.
`system dir *.io`	Executes the system "dir *.io" command
`system dir *.io del he.io exit`	Leaves the Simulator temporarily. Executes the system "dir *.io" and "del he.io" commands. Returns to the Simulator when the system "exit" command is executed.
`system -c del e:\temp\*.lod`	Deletes the specified temporary files and continue without issuing the continuation prompt.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.34 trace - Trace Through DSP Program

TRACE [count] [**CY**(cycles)/**LI**(lines)/**IN**(instructions)] [**H**(halt at breakpoints)]

The **trace** command gives a snap shot of the enabled registers and memory after each instruction or clock cycle is executed. Execution terminates after count number of cycles, lines, or instructions.

[count] Optional. When included, the count parameter indicates the number of cycles/lines/instructions to execute before halting execution. If no count is specified, the default count is 1.

[**CY**(cycles)/**LI**(lines)/**IN**(instructions)]

Optional. As the default, the **trace** command steps in instruction increments if viewing the assembly or register screens, and in source line increments if viewing the source screen. The **CY**, **LI** and **IN** options permit cycles, source line or instruction increments to be specified explicitly.

[**H**(halt at breakpoints)] Optional. The default is to ignore all breakpoints while the **trace** command is executing. The h parameter enables halting at breakpoints.

Table 12-37. TRACE Commands

Command	Resulting Simulator Action
trace	Executes one instruction or source line, depending on the display mode, then stops and displays the enabled registers and memory blocks.
trace li	Executes one source line, regardless of the display mode, then stops and displays the enabled registers and memory blocks.
trace 20	Executes 20 instructions or source lines, depending on the display mode, and displays the enabled registers and memory blocks after each trace execution. Ignores breakpoints.
trace 20 in	Executes 20 instructions, regardless of the display mode, and displays the enabled registers and memory blocks after each instruction. Ignores breakpoints.
trace 20 h	Executes 20 instructions and displays the enabled registers and memory blocks after each instruction. Halts if a breakpoint is encountered.
trace 10 cy	Executes 10 clock cycles and displays the enabled registers and memory blocks after each clock cycle. Ignores breakpoints.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.35 type - Display the Result Type of C Expression

TYPE **{**c_expression**}**

The **type** command is used to display the result type of a C expression in the Session window.

{c_expression**}** The C expression to be examined (remember to enclose in curly brackets). If the result of the expression is a storage location (e.g. just a variable name, or an element of an array), the address of the storage location, in addition to its data type will be displayed.

Table 12-38. TYPE Commands

Command	Resulting Simulator Action
type {count}	Displays the type and location of the variable count.
type {0.5+i}	Displays the type of the given expression: 0.5+i.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.36 unlock - Unlock Password Protected Device Type

UNLOCK dev_type password

The **unlock** command provides unlocks unannounced device types. Once unlocked, the device type may be selected for simulation using **device** command.

dev_type — The type of device that is to be unlocked.

password — The password to unlock the device.

Table 12-39. UNLOCK Commands

Command	Resulting Simulator Action
unlock 56001 x51-234	Enables device type 56001 for simulation using the password x51-234.

12.37 until - Execute Until Address

UNTIL addr [**H**(halt at breakpoints)]

The **until** command sets a temporary breakpoint at the specified line or address, then begins execution from the address indicated by the current program counter until that breakpoint is encountered. The Simulator then clears the temporary breakpoint and displays the enabled registers and memory blocks in the same manner as the **step** command.

addr — The addr parameter specifies the address in memory at which execution will be halted. The addr parameter may also be expressed as a line number in the program source file. Specification of a line number is valid only if the symbolic debug information has been loaded from a COFF format .cld file. (The debug information is generated at the time of assembly using the assembler's –g option.) Line numbers may be specified as filename@line_number for a line number in a particular file or simply by line_number for line numbers in the currently displayed file.

[**H**(halt at breakpoints)] — Optional. The default is to ignore all breakpoints while the **until** command is executing. The h parameter enables halting at breakpoints.

Table 12-40. UNTIL Commands

Command	Resulting Simulator Action
until 20	Executes until the instruction associated with line 20 in the current file is reached.
until p:$50	Executes until the instruction at hexadecimal address p:50 is reached. Ignores breakpoints.
until p:$50 h	Executes until the instruction at hexadecimal address p:50 is reached. Does not ignore breakpoints.
until lab_2	Executes until the instruction at label **lab_2** is reached.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.38 up - Move Up the C Function Call Stack

UP [n]

The **up** command is used to move up the call stack. It can be used in conjunction with the **where**, **frame**, and **down** commands to display and traverse the C function call stack.

After entering a new call stack frame using **up**, that call stack frame becomes the current scope for evaluation. This means that for C expressions, the **evaulate** command acts as though this new frame is the proper place to start looking for variables.

[n] Optional. The number of frames to move up. If no number is specified, the Simulator moves up one frame in the call stack.

Table 12-41. UP Commands

Command	Resulting Simulator Action
up	Moves up the call stack by one stack frame.
up 3	Moves up the call stack by three stack frames

12.39 view - Select Display Mode

VIEW [**A**(assembly)/**S**(source)/**R**(register)]

The **view** command changes the Simulator display mode. There are three display modes: assembly, source and register. section 1.7, “Setting Up the Display Environment,” describes the display environment. When no parameter is entered, the display mode cycles to the next display mode in the order: source - assembly - register. The same results can be obtained by typing ctrl-w.

[**A**(assembly)/**S**(source)/**R**(register)]

Optional. The mode that is to be displayed in the Session window. The assembly and source views will indicate the next instruction to be executed with a pointer to the left of the instruction. The register mode will display the values of registers enabled for display (see the **display** command), including highlighted values for registers that were written to in the last executed instruction.

Table 12-42. VIEW Commands

Command	Resulting Simulator Action
view	Selects the next display mode among source, assembly and register modes.
view s	Selects source display mode.
view a	Selects assembly display mode.
view r	Selects register display mode.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.40 wait - Wait Specified Time

WAIT [count(seconds)]

The **wait** command pauses for count seconds or until you click on **Cancel** before continuing to the next command. This command is particularly useful in command macros. If the **wait** command is entered without a count parameter, the pause will continue indefinitely and will only terminate if you click on **Cancel**.

[count(seconds)] Optional. The number of seconds to pause.

12.41 wasm - GUI Assembly window

WASM [**OFF**]

The **wasm** command opens an assembly window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

[**OFF**] Optional. Closes the assembly window.

Table 12-43. WASM Commands

Command	Resulting Simulator Action
wasm	Opens the assembly window for the current device.
wasm off	Closes the assembly window for the current device.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.42 watch - Set, Modify, View, or Clear Watch item

WATCH [#wn] [radix] reg/addr/expression/**{**c_expression}

WATCH [#wn] **OFF**

The **watch** command is used to add, modify, view, and clear items on the watch list. The watch list is a list of registers, addresses, and expressions that gets updated every time execution is halted or a breakpoint condition is met. If the **watch** command is performed without any parameters, the current watch list is displayed in the Session window.

[#wn] — Optional. An arbitrary number assigned to each watch list. Multiple lists can be defined, each being displayed in a separate window. If no number is specified, the Simulator will automatically add the watch item to the lowest numbered watch list.

[radix] — Optional. The radix parameter indicates the number base in which to display the watch item. The radix may be specified by using **D** for decimal, **F** for fractional, **H** for hexadecimal, or **U** for unsigned. If no radix is specified, the default radix of the specified item is used. The default display radix of a watch item can be changed with the **radix** command.

reg/addr/expression/{c_expression} — This parameter specifies the watch item to be added to the watch list. A watch item can be a register, address, expression, or C expression.

OFF — Used with the second form of the watch command. Turns off all watched registers, addresses, and expressions.

Table 12-44. WATCH commands

Command	Resulting Simulator Action
watch r0	Adds register r0 to the watch list.
watch x:0	Adds x:0 to the watch list.
watch {(count+1)%total}	Adds the given C expression to the watch list.
watch h {count/2}	Adds the given C expression to the watch list, in display radix hex.
watch b {flag}	Adds the given C variable to the watch list, in display radix binary.
watch r0+x:0	Adds the expression r0+x:0 to the watch list.
watch	Displays the watch list.
watch #3 off	Removes item number three from the watch list.
watch off	Removes all items from the watch list.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.43 wbreakpoint - GUI Breakpoint Window

WBREAKPOINT [**OFF**]

The **wbreakpoint** command opens a breakpoint window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

[**OFF**] Optional. Closes the wbreakpoint window.

Table 12-45. WBREAKPOINT Commands

Command	Resulting Simulator Action
wbreakpoint	Opens a breakpoint window for the current device.
wbreakpoint off	Closes the breakpoint window for the current device.

12.44 wcalls - GUI C Calls Stack window

WCALLS [**OFF**]

The **wcalls** command opens a C call stack window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

[**OFF**] Optional. Closes the stack window.

Table 12-46. WCALLS Commands

Command	Resulting Simulator Action
wcalls	Opens a C call stack window for the current device.
wcalls off	Closes the C call stack window for the current device.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.45 wcommand - GUI Command window

WCOMMAND [**OFF**]

The **wcommand** command opens a Command window. Only one Command window may be open. The current device is always the device that is related to the Command window. Use the **device** command to change the current device.

[**OFF**] Optional. Closes the command window.

Table 12-47. WCOMMAND Commands

Command	Resulting Simulator Action
wcommand	Opens a command window.
wcommand off	Closes the command window for the current device.

12.46 where - GUI C Calls Stack window

WHERE [[+/-]n]

The **where** command displays the C function calls in the Session window. It can be used in conjunction with the **frame**, **down**, and **up** commands to display and traverse the C function call stack.

[[**+/-**]n] Optional. This parameter indicates the number of frames of the call stack to display, where n is the number of frames. Optional, the + (plus) sign and the - (minus) sign can be used to indicate the whether to display the call stack beginning with the innermost or outermost frames. The + (plus) sign indicates innermost frames. The - (minus) sign indicates the outermost frames. If no sign is specified, the Simulator will assume that the innermost frames are intended for display.

Table 12-48. WHERE Commands

Command	Resulting Simulator Action
where	Display the call stack.
where 3	Display the three innermost frames in the call stack.
where -5	Display the five outermost frames in the call stack.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.47 winput - GUI File Input window

WINPUT [**OFF**]

The **winput** command opens an input window, listing all of the assigned inputs. See the **input** command for more information. Multiple input windows may be opened for separate devices when debugging on systems with multiple DSPs.

[**OFF**] Optional. Closes the winput window.

Table 12-49. WINPUT Commands

Command	Resulting Simulator Action
winput	Opens an input window for the current device.
winput off	Closes the input window for the current device.

12.48 wlist - GUI list window

WLIST [OFF]

The **wlist** command opens a list window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

[**OFF**] Optional. Closes the list window.

Table 12-50. WLIST Commands

Commands	Resulting Simulator Action
wlist lfile.1st	Opens a list window with the text file lfile.1st displayed.
wlist win2 lfile.1st	Opens a list window with a window number of 2 with text lfile.txt displayed. If list window 2 already exists, replaces the contents with lfile.1st.
wlist win2 off	Closes list window number 2.
wlist off	Closes all open list windows.
wlist win3	Opens a list window with a window number of 3 with no text file displayed.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.49 wmemory - GUI Memory window

WMemory [#wn] space [addr]

WMemory [#wn] [**OFF**]

The **wmemory** command opens a memory window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

[#wn]	Optional. Window number. The window number is an arbitrarily assigned number, used to keep track of possible multiple memory windows. If no window number is specified when opening a new window, the Simulator will automatically assign the next available number.
space	The space parameter indicates the memory space to display.
[addr]	Optional. Used with the first form of the **wmemory** command. Indicates the address in memory at which to begin displaying values.
[**OFF**]	Optional. Closes the memory window. If no window number (#wn) is specified when closing a window, all memory windows will be closed.

Table 12-51. WMEMORY Command

Command	Resulting Simulator Action
wmemory pi	Opens a memory window for the internal program (pi) memory space for the current device.
wmemory xi 0	Opens a memory window for the xi memory space containing address 0 for the current device.
wmemory win3 x	Opens a memory window for memory space x with a window number of 3 for the current device.
wmemory off	Closes all memory windows for the current device.
wmemory win3 off	Closes memory window 3 for the current device.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.50 woutput - GUI File Output window

WOUTPUT [**OFF**]

The **woutput** command opens a file output window, listing all of the assigned outputs. See the **output** command for more information. Multiple output windows may be opened for each separate device when debugging on systems with multiple DSPs.

[**OFF**] Optional. Closes the output window.

Table 12-52. WOUTPUT Commands

Command	Resulting Simulator Action
woutput	Opens an output window for the current device.
woutput off	Closes the output window for the current device.

12.51 wregister - GUI Register window

WREGISTER [**win**n] [periph/**OFF**]

The **wregister** command opens a register window. Multiple register windows may be opened for each peripheral or for separate devices when debugging on systems with multiple DSPs.

[**win**n] Optional. Window number. The window number is an arbitrarily assigned number, used to keep track of multiple register windows. If no window number is specified when opening a new window, the Simulator will automatically assign the next available number.

[periph/**OFF**] Optional. The periph parameter is the name of a peripheral and indicates that only the registers of that particular peripheral should be displayed. If no peripheral name is specified, the Simulator will display the core registers.

The **OFF** parameter closes the register window. If no window number (**win**n) is specified when closing a window, all register windows will be closed.

Table 12-53. WREGISTER Commands

Command	Resulting Simulator Action
wregister	Opens a register window for the current device.
wregister win3 host	Opens a register window with a window number of 3, and displays only the registers associated with the host peripheral for the current device.
wregister off	Closes all register windows for the current device.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

12.52 wsession - GUI Session window

WSESSION [**OFF**]

The **wsession** command opens a Session window. Only one Session window may be open at a time. The current device is always the device that is displayed in the Session window. Use the **device** command to change the current device.

[**OFF**] Optional. Closes the Session window.

Table 12-54. WSESSION Commands

Command	Resulting Simulator Action
wsession	Opens a session window for the current device
wsession off	Closes the session window.

12.53 wsource - GUI Source window

WSOURCE [**OFF**]

The **wsource** command opens a source code window which list the source code of the currently loaded program (.cld file). Source debug information must be included in the source file for source code to appear in this window.

[**OFF**] Optional. Closes the source window.

Table 12-55. WSOURCE Commands

Command	Resulting Simulator Action
wsource	Opens a source window for the current device.
wsource off	Closes the source windows for the current device.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

12.54 wstack - GUI Stack window

WSTACK [**OFF**]

The **wstack** command opens a device stack window, which displays the current call stack. Multiple stack windows may be opened for debugging on systems with multiple DSPs.

[**OFF**] Optional. Closes the stack window.

Table 12-56. WSTACK Commands

Command	Resulting Simulator Action
wstack	Opens a stack window for the current device.
wstack off	Closes the stack window for the current device.

12.55 wwatch - GUI Watch window

WWATCH [**win**_num] [#n] [radix] reg/addr/expression/{c_expression}

WWATCH [**win**_num] [#n] [**OFF**]

The **wwatch** command opens a watch window. Multiple watch windows may be opened - one for each watch list or for separate devices when debugging on systems with multiple DSPs.

[**win**_num] Optional. An arbitrary number assigned to each watch window. Multiple lists can be defined, each being displayed in a separate window. If no number is specified, the Simulator will automatically add the watch item to the lowest numbered watch window.

[#n] Optional. Watch item number. If no number is specified, the Simulator will assign the next available number to the watch item.

[radix] Optional. The radix parameter indicates the number base in which to display the watch item. The radix may be specified by using **d** for decimal, **f** for fractional, **h** for hexadecimal, or **u** for unsigned. If no radix is specified, the default radix of the specified item is used. The default display radix of a watch item can be changed with the **radix** command.

reg/addr/expression/{c_expression}

This parameter specifies the watch item to be added to the watch list. A watch item can be a register, address, expression, or C expression.

[**OFF**] Optional. Closes the specified watch window. If no window number is specified, closes all watch windows.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Table 12-57. WWATCH commands

Command	Resulting Simulator Action
`wwatch r0`	Opens a watch window for the current device with the register r0 displayed. If the window already exists, adds r0 to the list of watched items.
`wwatch x:$100`	Opens a watch window for the current device with the memory location x:$100 displayed. If the window already exists, adds x:$100 to the list of watched items.
`wwatch r2+3`	Opens a watch window for the current device with the expression r2+3 displayed. If the window already exists, adds r2+3 to the list of watched items.
`wwatch win2 r0`	Opens a watch window for the current device with a window number of 2 with the register r0 displayed. If the window already exists, adds r0 to the list of watched items.
`wwatch off`	Closes all watch windows for the current device.
`wwatch win3 off`	Closes watch window 3 for the current device.
`wwatch #2 off`	Removes watch element #2 from first watch window's list of watched elements.
`wwatch win4 @2 off`	Removes watch element #2 from watch window 4's list of watched elements.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Chapter 13
C Library Functions

The SIMDSP Simulator package includes several object code libraries of Simulator functions that were used to create the Simulator. The libraries allow the user to build his own customized Simulator and integrate it with your unique project.

The Simulator package includes the source code for many of the Simulator functions, including the code that defines the dsp external memory accesses, the code for the main entry point, the code for the terminal I/O functions, and example code for a non-display version of the Simulator. The source code can be modified to create a Simulator customized for a particular application.

Object libraries are supplied which support display or non-display versions of the Simulator. The user may choose to eliminate the user interface functions altogether and control the simulation directly through lower level function calls.

The rest of this chapter covers these aspects plus the following aspects of the specification and use of the libraries:

- Section 13.2, "Simulator Object Library Entry Points," lists each Simulator entry point that is available to you the user.
- Section 13.3, "Simulator External Memory Functions," provides a description of the external memory access functions.
- Section 13.4, "Simulator Screen Management Functions," provides a description of the terminal I/O functions.
- Section 13.5, "Non-Display Simulator," Topics concerning the non-display version of the Simulator are discussed in section 13.5.
- Section 13.6, "Multiple Device Simulation," Simulation of multiple dsp devices is fully supported by the dsp library functions.
- Section 13.6, "Multiple Device Simulation," discusses topics related to simulating and interconnecting multiple dsp devices.
- Section 13.7, "Reserved Function Names," provides a description of the public function names used by the Simulator.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

- Section 13.8, "Simulator Global Variables," describes the global variables used by the Simulator.
- Section 13.9, "Modification of Simulator Global Structures," describes modifications that can be made to the Simulator global structures.
- Section 13.10, "Modification of Device Global Structures," describes modifications that can be made to device global structures.

13.1 C Object Libraries

The Simulator software includes object libraries that enable you to rebuild the Simulator. A separate set of display and non-display libraries are provided, so that you have the option of generating a non-display version of the Simulator. The libraries with the prefix “ww”, followed by the family device number, contain the display version of the object modules. The libraries with the prefix “nw”, followed by the family device number, contain the non-display versions of the same object modules. In addition, the libraries with the prefix “cm”, followed by the family device number, contain object modules required by both the display and non-display versions of the Simulator. As an example, relinking the display version of the sim56100 Simulator requires libraries ww56100 and cm56100; a non-display version of the Simulator requires the libraries nw56100 and cm56100.

13.2 Simulator Object Library Entry Points

The following is a quick reference list of the higher level Simulator entry points provided in the Simulator object libraries. The prefix indicates whether or not the function is available in the non-display version of the Simulator. Function names beginning with the prefix `dsp_` or `dspt_` are available to both the display and non-display versions of the Simulator, while function names beginning with `sim_` are only available when using a display version of the Simulator. The `dspt_` prefix indicates a device dependent function. The `_xxxxx` suffix on these indicate a device family number. Lower level Simulator functions, which have a prefix of `dspl_`, `siml_` or `dsptl_`, are not intended for direct access by the user’s program. They are not described in this document. The higher level functions listed below are described in detail in Section 13.2.1, "dspt_masm_xxxxx: Assemble DSP Mnemonic," through Section 13.2.30, "sim_gtcmd: Get Command String from Terminal," . Table 13-1 lists the higher level functions.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 13-1. Higher Level Functions

Function	Description
`dspt_masm_xxxxx(mnemonic,ops,err);`	Assemble mnemonic string to ops
`dspt_unasm_xxxxx(ops,sr,omr,sdbp);`	Disassemble dsp opcodes
`dsp_exec(devn);`	Execute one clock cycle for dsp device
`dsp_findmem(devn,memname,map);`	Get map index for memory prefix
`dsp_findpin(devn,pinname,pinnum);`	Get pin number for pin name
`dsp_findport(devn,portname,pnum,pmask);`	Get port number and mask for port name
`dsp_findreg(devn,regname,pval,rval);`	Get peripheral and register index for register
`dsp_fmem(devn,map,addr,blocksz,val);`	Fill memory block with a value
`dsp_free(devn);`	Free memory allocated for a dsp device
`dsp_init(devn,mode);`	Initialize selected device and mode
`dsp_ldmem(devn,filename);`	Load device memory from filename
`dsp_load(filename);`	Load all device states from filename
`dsp_new(devn,device_type);`	Create new dsp device
`dsp_path(path,base,suffix,new_name);`	Create filename from path, base and suffix
`dsp_rapin(devn,pin_number,val);`	Read output analog pin state from device
`dsp_rmem(devn,map,addr,mem_val);`	Read dsp memory map address to mem_val
`dsp_rpin(devn,pin_number);`	Read output pin state from device
`dsp_rport(devn,port,data,force);`	Read output port state from device
`dsp_rreg(devn,periphn,regn,regval);`	Read dsp peripheral register to regval
`dsp_save(filename);`	Save the state of all devices to filename
`dsp_startup();`	Initialize Simulator structures
`dsp_unlock(device_type,password);`	Unlock password protected device type

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Table 13-1. Higher Level Functions

Function	Description
`dsp_wapin(devn,pin,value);`	Write device analog input pin with value
`dsp_wmem(devn,map,addr,val);`	Write dsp memory map address with val
`dsp_wpin(devn,pin,value);`	Write device input pin with value
`dsp_wport(devn,port,mask,data,force);`	Write device port with data and force value
`dsp_wreg(devn,periphn,regn,regval);`	Write dsp peripheral register with regval
`sim_docmd(devn,command_string);`	Perform Simulator command on dsp device
`sim_gmcmd(devn,command_string);`	Get command string from macro file
`sim_gtcmd(devn,command_string);`	Get command string from terminal

13.2.1 dspt_masm_xxxxx: Assemble DSP Mnemonic

```
int dspt_masm_xxxxx(mnemonic,ops,error_ptr)
char *mnemonic;         /* Pointer to assembler mnemonic string */
unsigned long *ops;     /* Where to put the words of assembled opcode */
char **error_ptr;       /* Will point to error message if an error occurs */
```

This function invokes the single line assembler to assemble a DSP mnemonic. Table 13-2 list the different integer code it returns:

Table 13-2. Integer Code

Code	Description
-1	An error occurred. The user supplied error pointer will point to a message that explains the error.
0	The line mnemonic provided was a comment
1	The mnemonic assembled correctly and required 1 word of code. The code will be in the ops[0] location.
2	The mnemonic assembled correctly and required 2 words of code. The first word will be in placed in ops[0], the second in ops[1].
3	The mnemonic assembled correctly and required 3 words of code. The first word will be in placed in ops[0], the second in ops[1], the third in ops[2].

Note: Note that the xxxxx in the function name should be replaced by a device family number. It should be 56k for the 56000 family devices, 56100 for the 56100 family devices, and 96k for the 96000 family devices.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Example 13-1 shows how to use the dspt_masm_xxxxx function.

Example 13-1. dspt_masm_xxxxx

```
/* Assemble the instruction "move r0,r1" */
unsigned long opcodes[3];
char *error_ptr;
int retval;
retval=dspt_masm_56k("move r0,r1",&opcodes[0],&error_ptr);
```

13.2.2 dspt_unasm_xxxxx: Disassemble DSP Mnemonics

int dspt_unasm_xxxxx(ops,return_string,sr,omr,gdbp)

unsigned long *ops; /* Pointer to opcodes to be disassembled */

unsigned long sr; /* Value of device status register */

unsigned long omr; /* Value of device operating mode register */

char *gdbp; /* Return value reserved for use by debugger*/

char *return_string; /* Pointer to return character buffer */

This function disassembles ops[0] (and possibly ops[1] and ops[2] if ops [0] requires a second or third word) and places the disassembled mnemonic in the return_string buffer supplied by the user. If correct disassembly requires a device status register and/or operating mode register value, the values should be provided in the sr and omr parameters. The gdbp parameter is a pointer reserved for use by the symbolic debugger, and should be NULL for other applications.

The mnemonic may require as many as 120 characters of return buffer. The function returns the number (1 to 3) of words consumed by the disassembly. It returns 0 for illegal opcodes and a return string containing a DC directive. See Example 13-2.

Note: The xxxxx in the function name should be replaced by a device family number. It should be 56k for the 56000 family devices, 56100 for the 56100 family devices, and 96k for the 96000 family devices.

Example 13-2. dspt_unasm_xxxxx

```
/* Disassembly of the opcode representing NOP */
unsigned long ops[3];   /*Instruction words to be disassembled.*/
char return_string[120];/*The return mnemonic goes here.*/
int numwords;           /*Number of operands used by disassembler.*/
ops[0]=0L;
ops[1]=0L;
ops[2]=0L;
numwords=dspt_unasm_56k(ops,return_string,0L,0L,NULL);
/* Now numwords==1, return_string=="nop"*/
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.2.3 dsp_exec: Execute Single Device Clock Cycle

```
dsp_exec(device_index)
int device_index;       /* DSP device index to be affected by command */
```

This function executes a single dsp device cycle and updates the selected dsp device structure. All device inputs, outputs and registers are updated as a result of this call. In addition, it tests user-defined breakpoint conditions and clears the device's executing status flag if a breakpoint occurs. It also calls the functions which handle cycle by cycle I/O from assigned input or output files. See Example 13-3.

Example 13-3. dsp_exec

```
/*Execute 1000 cycles on a device*/
int devn;
int cycles;
dsp_startup();
devn=0;
dsp_new(devn,"56116";         /* allocate new device */
for(cycles=0;cycles<1000;cycles++) dsp_exec(devn);
```

13.2.4 dsp_findmem: Get Map Index for Memory Prefix

```
dsp_findmem(device_index,memory_name,memory_map)
int device_index;             /* DSP device index to be affected by command */
char *memory_name;            /* memory space name */
enum memory_map *memory_map;  /* return memory map type */
```

This function searches the `dt_var.mem` structure for a match to the `memory_name` string provided in the function call. If a match is found, `dsp_findmem` returns the memory map maintype structure value through the `memory_map` parameter and 1 as the function return value; otherwise it just returns 0 as the function return value. See Example 13-4.

For a list of memory names use the Simulator **help mem** command.

Example 13-4. dsp_findmem

```
#include "coreaddr.h"
int devn;
enum memory_map map;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
ok=dsp_findmem(devn,"p",&map) /* Get memory map index for "p" memory */
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.2.5 dsp_findpin: Get Pin Number for Pin Name

```
dsp_findpin(device_index,pin_name,pin_number)
int device_index;       /* DSP device index to be affected by command */
char *pin_name;         /* pin name */
int *pin_number;        /* return pin index */
```

This function searches the dt_var.xpin structures for a match to the pin_name string provided in the function call. If a match is found, dsp_findpin returns the pin number through the pin_number parameter and 1 as the function return value; otherwise it just returns 0 as the function return value. See Example 13-5.

Use the Simulator's **help pin** command to produce a list of the valid pin names.

Example 13-5. dsp_findpin

```
int devn;
int pinnum;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
ok=dsp_findpin(devn,"reset",&pinnum);/* Get index for "reset" pin */
```

13.2.6 dsp_findport: Get Port Number and Mask for Port Name

```
dsp_findport(device_index,port_name,port_number,mask_val)
int device_index;       /* DSP device index to be affected by command */
char *port_name;        /* port or peripheral name */
int *port_number;       /* return memory map index */
unsigned long *mask_val;/* Pin mask for this port or peripheral name */
```

This function searches the dt_var.xport structure and the dt_var.periph structures for a match to the port_name string provided in the function call. If a match is found, dsp_findport returns the port number through the port_number parameter, the port mask value through the mask_val parameter, and 1 as the function return value; otherwise it just returns 0 as the function return value.

The Simulator "help port" and **help periph** commands may also be used to produce a list of valid port and peripheral information. See Example 13-6.

Example 13-6. dsp_findport

```
int devn;
int pnum;
unsigned long pmask;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
ok=dsp_findport(devn,"portb",&pnum,&pmask);/* Get info for "portb" */
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.2.7 dsp_findreg: Get Peripheral and Register Index for Register Name

```
dsp_findreg(device_index,reg_name,periph_number,reg_number)
int device_index;       /* DSP device index to be affected by
command*/

char *reg_name;         /* register name */
int *periph_number;     /* return peripheral index */
int *reg_number;        /* return register index */
```

This function searches the dt_var.periph structures for a match to the reg_name string provided in the function call. If a match is found, dsp_findreg returns the peripheral index through periph_number, the register number through the reg_number parameter and 1 as the function return value; otherwise it just returns 0 as the function return value.

You may also use the Simulator **help reg** command to obtain a list of the valid periph_num and reg_num values, and reg_val size for each register. See Example 13-7.

Example 13-7. dsp_findrg

```
int devn;
int regnum;
int pnum;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a
56116 */

ok=dsp_findreg(devn,"pc",&pnum,&regnum);/* Get index for "pc" register */
```

13.2.8 dsp_free: Free a Device Structure

```
dsp_free(device_index)
int device_index;       /* DSP device index to be affected by command */
```

This function frees all allocated memory associated with a device structure and closes any open files associated with the device structure. See Example 13-8.

Example 13-8. dsp_free

```
/* Create three new device structures, then get rid of device 2. */
dsp_startup();
dsp_new(0,"56116");           /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");           /* Allocate structure for device 1, a 56116 */
dsp_new(2,"56116");           /* Allocate structure for device 2, a 56116 */
dsp_free(1);                  /* Free structure previously allocated for
device 1 */
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.2.9 dsp_fmem: Fill Memory Block with a Value

```
dsp_fmem(device_index,memory_map,address,block_size,value)
int device_index;             /* DSP device index to be affected by command */
enum memory_map memory_map;   /* memory designator */
unsigned long address;        /* DSP memory start address to write */
unsigned long block_size;     /* Number of locations to write */
unsigned long *value;         /* Pointer to value to write to memory location
*/
```

This function writes a memory block of selected dsp memory.

The memory_map parameter is a memory type that selects the appropriate dt_memory structure from dt_var.mem for the selected device. These structures are describe in the simdev.h file which is included with the Simulator. The memory_map parameter can be obtained with the function dsp_findmem by using the memory name as a key. Use the Simulator **help mem** command for a list of valid memory names. The memory_map enum is memory_map_ concatenated with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

If the selected memory map requires two word values, the least significant word should be at the value location and the most significant word at the value+1 location.

If the memory address selects an external memory location, the dspl_xmwr function will be called. The dspl_xmwr function is provided in source form in the file simvmem.c and can be modified to simulate special external memory characteristics. See Example 13-9.

Example 13-9. dsp_fmem

```
/* Write 300 locations beginning at p:$200 with the value 4 */
int devn;
unsigned long address, memval, blocksize;
address=0x200L;
blocksize=300;
memval=4L;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
dsp_fmem(devn,memory_map_p,address,blocksize,&memval);
```

13.2.10 dsp_init: Initialize a Single DSP Device Structure

```
dsp_init(device_index)
int device_index;       /* DSP device index to be affected by command */
```

This function initializes a device to the same state that existed following the dsp_new() call which created it. It is equivalent to performing the Simulator **reset s** command. All memory spaces are cleared, the registers are reset, breakpoints and input/output file assignments are cleared. See Example 13-10.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Example 13-10. dsp_init

```
dsp_startup();
dsp_new(0,"56116");          /* Create new dsp structure */
.
.                            /* Other Simulator commands */
.
dsp_init(0);                 /* Reinitialize device 0 */
```

13.2.11 dsp_ldmem: Load DSP Memory from OMF File

```
int dsp_ldmem(device_index,filename)
int device_index;       /* DSP device index to be affected by command */
char *filename;         /* Full pathname of OMF format file to be loaded */
```

This function loads the memory space of a specified dsp device from an object file. The file may be created as the output from the DSP MACRO ASSEMBLER, or by using the Simulator **save** command, and may be either COFF format or " .lod" format. In order to specify a COFF format file, the filename suffix must be " .cld". A filename with any other suffix is assumed to be in ".lod" format.

This is a lower level function that does not invoke the user interface modules for pathname and automatic .lod suffix extension. The entire pathname must be specified. The function returns 1 if the load is successful, 0 if an error occurred loading the file. See Example 13-11

Example 13-11. dsp_Idmem

```
/* Create DSP device structures for a three device simulation. */
int devn;
int err;
dsp_startup();
for (devn=0;devn<3;devn++)
dsp_new(devn,"56116");        /* Create new dsp structures */
/* Load device 1 with a program named filter2.lod.*/
err=dsp_ldmem(1,"filter2.lod");
```

13.2.12 dsp_load: Load All DSP Structures from State File

```
int dsp_load(filename)
char *filename;         /* Full name of State File to be loaded */
```

This function loads the Simulator state of all devices from a specified Simulator state file. It is not necessary to allocate the device structures prior to calling dsp_load. This function does not invoke the user interface modules for pathname and automatic .sim suffix extension; the entire filename must be specified. See Example 13-12.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Example 13-12. dsp_load

```
int err;
dsp_startup();
err=dsp_load("lunchbrk.sim");
```

13.2.13 dsp_new: Create New DSP Device Structure

```
dsp_new(device_index,device_type)
int device_index;       /* DSP device index to be affected by command */
char *device_type;      /* Name corresponding to DSP device type */
```

This function creates a new dsp structure that represents a DSP device and initializes it. It will be necessary to use the dsp_unlock( ) function call prior to dsp_new( ) if the selected device type is password protected. See Example 13-13.

Example 13-13. dsp_new

```
/* Create DSP device structures for a three device simulation. */
int devn;
dsp_startup();
for (devn=0;devn<3;devn++)
dsp_new(devn,"56116");        /* Create new dsp structures */
```

13.2.14 dsp_path: Construct Filename

```
dsp_path(path_name,base_name,suffix,new_name)
char *path_name;        /* Directory pathname */
char *base_name;        /* Base filename to be appended to path_name */
char *suffix;           /* Suffix string to be appended to base_name */
char *new_name;         /* Pointer to return buffer for constructed pathname
*/
```

This function concatenates the user-provided pathname, base name and suffix. If the base_name begins with a pathname separator or with a device designator, the path_name will not be prepended to the base_name. If the base_name already ends with '.' and some suffix, the suffix will not be appended. See Example 13-14.

Example 13-14. dsp_path

```
/* Load a file named filter2.lod from the current working directory for device
0. */
#include "simcom.h"
#include "simdev.h"
extern struct dev_const dv_const;/* Simulator device structures */
char newfn[80];
dsp_startup();
dsp_new(0,"56116");           /* Create new dsp structure */
dsp_path(dv_const.sv[0]->pathwork,"filter2","lod",newfn);
dsp_ldmem(0,newfn);           /* Load file into dsp device 0 */
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.2.15 dsp_rapin: Read DSP Analog Pin State

```
int dsp_rapin(device_index,pin,retvf)
int device_index;       /* DSP device index to be affected by command */
int pin;                /* Pin number to read */
float *retvf;           /* Pointer to floating point (single precision) return
value */
```

This function reads a dsp analog device pin value. It is only valid for device pins which are defined as having analog values, such as codec output pins; other pins will return 0.0 as the analog value. The function return value will be DSP_PINVAL_L and a floating point value returned in retvf if found; or DSP_ERR if there is an error condition. Use the Simulator's **help pin** command to produce a list of the valid pin names and each pin's corresponding pin index. The device pin number can also be obtained by using the pin name as a key and calling the function dsp_findpin. The DSP_PINVAL return values are defined in the simdev.h file. See Example 13-15.

Example 13-15. dsp_rapin

```
int devn;
int pinnum;
int err;
float aval;
dsp_startup();
devn=0;
dsp_new(devn,"56156");        /* Allocate structure for device 0, a 56156 */
dsp_findpin(devn,"spkp",&pinnum);   /* Get pin number for pin named spkp */
err=dsp_rapin(devn,pinnum,&aval);   /* Read value of device 0 pin spkp*/
```

13.2.16 dsp_rmem: Read DSP Memory Location

```
int dsp_rmem(device_index,memory_map,address,return_value)
int device_index;             /* DSP device index to be affected by command */
enum memory_map memory_map;   /* memory designator */
unsigned long address;        /* DSP memory address to read */
unsigned long *return_value;  /* Memory value (or values) will be returned
here */
```

This function reads the contents of a selected dsp memory location and writes it to return_value. If the memory_map implies a two word value, the least significant word will be returned to return_value; the most significant word will be returned to the return_value+1 location. This function also returns a flag that indicates whether or not the memory location exists. It returns 1 if the location exists, 0 otherwise.

The memory_map parameter selects the appropriate dt_memory structure from dt_var.mem for the selected device. These structures are describe in the simdev.h file which is included with the Simulator. The memory_map parameter can be obtained with the function dsp_findmem by using the memory name as a key. Use the Simulator **help mem** command for a list of valid memory names. The memory_map enum is memory_map_ concatenated

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

This function calls the function dspl_xmrd() if the address indicates an external memory location. The dspl_xmrd() function is provided in source form in the file simvmem.c and can be modified to simulate special external memory characteristics. See Example 13-16.

Example 13-16. dsp_rmem

```
/* Read X memory location 100 from device 0. */
unsigned long address;
unsigned long memval;
int devn;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
address=100L;
ok=dsp_rmem(devn,memory_map_x,address,&memval);
```

13.2.17 dsp_rpin: Read DSP Pin State

```
int dsp_rpin(device_index,pin)
int device_index;       /* DSP device index to be affected by command */
int pin;                /* Pin number to read */
```

This function reads a dsp device pin value. The return value may be DSP_PINVAL_L, DSP_PINVAL_H, DSP_PINVAL_F, DSP_PINVAL_0, or DSP_PINVAL_1 indicating low output, high output, floating, low input or high input pin state. Use the Simulator's **help pin** command to produce a list of the valid pin names and each pin's corresponding pin index. The device pin number can also be obtained by using the pin name as a key and calling the function dsp_findpin. The DSP_PINVAL return values are defined in the simdev.h file. See Example 13-17.

Example 13-17. dsp_rpin

```
int devn;
int pinnum;
int pin_value;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
dsp_findpin(devn,"rw",&pinnum);/* Get pin number for pin named rw */
pin_value=dsp_rpin(devn,pinnum);/* Read value of device 0 pin rw */
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.2.18 dsp_rport: Read DSP Port State

```
dsp_rport(device_index,port,data,force)
int device_index;       /* DSP device index to be affected by command */
int port;               /* Port number to read */
unsigned long *data;    /* Return port data value goes here */
unsigned long *force;   /* Return port forcing state goes here */
```

This function reads a dsp device port state. It returns two values. The value returned in the `data` parameter contains the current pin data state for all pins in the port. In the case of input pins, this is the last value written to the input pin; in the case of output pins the data state is the last data written to the port by the device. The value returned in the `force` parameter indicates which port bits are actually being driven as outputs by the device.

The `port` parameter acts as the index to the `dev_var.xportval` array. A list of port names and the corresponding port index can be obtained using the Simulator's **help port** and **help periph** commands. The port index can also be determined by using the port name as a key when calling `dsp_findport`. See Example 13-18.

Example 13-18. dsp_rport

```
int devn;
int portnum;
unsigned long portbdata, portbforce;
unsigned long portmask;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
dsp_findport(devn,"portb",&portnum,&portmask);
dsp_rport(devn,portnum,&portbdata,&portbforce);
```

13.2.19 dsp_rreg: Read a DSP Device Register

```
dsp_rreg(device_index,periph_num,reg_num,reg_val)
int device_index;       /* DSP device index to be affected by command */
int periph_num;         /* DSP peripheral number */
int reg_num;            /* DSP register number */
unsigned long *reg_val; /* Return register value goes here */
```

This function reads a selected register from the regval array in a DSP dev_periph structure. Registers which return more than one word as the register value will return the least significant word in reg_val[0].

Use the Simulator **help reg** command to obtain a list of the valid `periph_num` and `reg_num` values, and `reg_val` size for each register. Also, `dsp_findreg` can be used to obtain the peripheral and register number by using the register name as a key. See Example 13-19.

Example 13-19. dsp_rreg

```
int devn;
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

```
int periph_num, reg_num;
unsigned long regval;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
if (dsp_findreg(devn,"pc",&periph_num,&reg_num))
dsp_rreg(devn,periph_num,reg_num,&regval);
```

13.2.20 dsp_save: Save All DSP Structures to State File

```
int dsp_save(filename)
char *filename;         /* Full name of State File to be saved */
```

This function saves a dsp device structure to a simulation state file. This function does not invoke the user interface functions which provide pathname and .sim suffix extension, so the entire filename must be specified. The function returns 1 if the save is successful, 0 if an error occurs when saving the file. This function will call the function dspl_xmsave as one of the steps of saving the dsp structure. See Example 13-20.

Example 13-20. dsp_save

```
int ok;
dsp_startup();
dsp_new(0,"56116");           /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");           /* Allocate structure for device 1, a 56116 */
                        /* Save device 0 and 1 to state file lunchbrk.sim. */
ok=dsp_save("lunchbrk.sim");
```

13.2.21 dsp_startup: Initialize DSP Structures

```
int dsp_startup();
```

This function initializes DSP structures. It should be called once (and only once) at the first of your program prior to any calls to dsp_new. See Example 13-21

Example 13-21. dsp_startup

```
dsp_startup();
dsp_new(0,"56116");           /* Allocate structure for device 0, a 56116 *
dsp_new(1,"56116");           /* Allocate structure for device 1, a 56116 *
```

13.2.22 dsp_unlock: Unlock Password Protected Device Type

```
dsp_unlock(device_type, password)
char *password;         /* Pointer to string containing password */
char *device_type;      /* Name corresponding to DSP device type */
```

This function provides the password for protected device types. It must be used prior to the dsp_new function call if the device type is password protected. See Example 13-22.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Example 13-22. dsp_unlock

```
/* Create a device simulation of the password protected 56001 device */
int devn;
dsp_startup();
dsp_unlock("56001","x51-234"); /* provide password for device */
devn=0;
dsp_new(devn,"56001");        /* Create new dsp structures */
```

13.2.23 dsp_wapin: Write DSP Analog Pin State

```
int dsp_wapin(device_index,pin,value)
int device_index;       /* DSP device index to be affected by command */
int pin;                /* Pin number to write*/
float value;            /* Input value for specified pin */
```

This function writes a selected dsp device pin with a single precision floating point input value. Use the Simulator's **help pin** command to produce a list of the valid pin names and each pin's corresponding pin index. The device pin number can also be obtained by using the pin name as a key and calling the function dsp_findpin. See Example 13-23.

Example 13-23. dsp_wapin

```
#include "simcom.h"
#include "simdev.h"
/* Write the reset pin of device 0 with a high level. */
int devn;
int pinnum;
float pinval;
dsp_startup();
devn=0;
dsp_new(devn,"56156");        /* Allocate structure for device 0, a 56156 */
dsp_findpin(devn,"mic",&pinnum);/* Get pin number for pin named mic */
pinval=0.709;
dsp_wapin(devn,pinnum,pinval);
```

13.2.24 dsp_wmem: Write DSP Memory Location

```
dsp_wmem(device_index,memory_map,address,value)
int device_index;             /* DSP device index to be affected by command */
enum memory_map memory_map;   /* memory designator */
unsigned long address;        /* DSP memory address to write */
unsigned long *value;         /* Pointer to value to write to memory location
*/
```

This function writes a selected dsp memory location.

The memory_map parameter is selects the appropriate dt_memory structure from dt_var.mem for the selected device. These structures are describe in the simdev.h file which is included with the Simulator. The memory_map parameter can be obtained with the function dsp_findmem by using the memory name as a key. Use the Simulator **help mem** command

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

for a list of valid memory names. The memory_map enum is memory_map_ concatenated with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

If the selected memory map requires two word values, the least significant word should be at the value location and the most significant word at the value+1 location.

If the memory address selects an external memory location, the dspl_xmwr function will be called. The dspl_xmwr function is provided in source form in the file simvmem.c and can be modified to simulate special external memory characteristics. See Example 13-24.

Example 13-24. dsp_wmem

```
int devn;
unsigned long address, memval;
address=200L;
memval=0L;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
dsp_wmem(devn,memory_map_p,address,&memval);
```

13.2.25 dsp_wpin: Write DSP Pin State

```
int dsp_wpin(device_index,pin,value)
int device_index;       /* DSP device index to be affected by command */
int pin;                /* Pin number to write*/
int value;              /* Input value for specified pin */
```

This function writes a selected dsp device pin with a value DSP_PINVAL_L, DSP_PINVAL_H, DSP_PINVAL_F, DSP_PINVAL_N, or DSP_PINVAL_P indicating low, high, floating, negative pulse, or positive pulse. Use the Simulator's **help pin** command to produce a list of the valid pin names and each pin's corresponding pin index. The device pin number can also be obtained by using the pin name as a key and calling the function dsp_findpin. The DSP_PINVAL values are defined in the simdev.h file. See Example 13-25.

Example 13-25. dsp_wpin

```
#include "simcom.h"
#include "simdev.h"
/* Write the reset pin of device 0 with a high level. */
int devn;
int pinnum;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
dsp_findpin(devn,"reset",&pinnum);/* Get pin number for pin named reset */
dsp_wpin(devn,pinnum,DSP_PINVAL_H);
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.2.26 dsp_wport: Write DSP Port State

```
dsp_wport(device_index,port,mask,data,force)
int device_index;       /* DSP device index to be affected by command */
int port;               /* Port number to write */
unsigned long mask;     /* Pin mask for this port   */
unsigned long data;     /* Port data value */
unsigned long force;    /* Port forcing state */
```

This function forces data on a dsp device port from outside the device. The value supplied in the data parameter contains the new input data to be written to the port. The value supplied in the force parameter indicates which port bits are actually being driven as inputs to the device. The value supplied in the mask parameter specifies which pins in the port are to be affected by this write; the other pins in the port remain in their previous state.

The port parameter acts as the index to the dev_var.xportval array. A list of port names and the corresponding port index can be obtained using the Simulator's **help port** and **help periph** commands. The port index and mask value can also be obtained by using the port name as a key when calling dsp_findport.

This function call can be paired with the dsp_rport function to simulate a port to port connection between devices. See Example 13-26.

Example 13-26. dsp_wport

```
/* Write portb of device 1 from portb of device 0 */
int portnum;
unsigned long portbdata, portbforce;
unsigned long portmask;
dsp_startup();
dsp_new(0,"56116");           /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");           /* Allocate structure for device 1, a 56116 */
dsp_findport(0,"portb",&portnum,&portmask);
dsp_rport(0,portnum,&portbdata,&portbforce)
dsp_wport(1,portnum,portmask,portbdata,portbforce)
```

13.2.27 dsp_wreg: Write a DSP Device Register

```
dsp_wreg(device_index,periph_num,reg_num,reg_val)
int device_index;       /* DSP device index to be affected by command */
int periph_num;         /* DSP peripheral number */
int reg_num;            /* DSP register number */
unsigned long *reg_val; /* Value to be written to register */
```

This function writes a selected register in the a dsp structure.

Use the Simulator **help reg** command to obtain a list of the valid periph_num and reg_num values, and reg_val size for each register. Also, the function dsp_findreg can be used to obtain the peripheral and register number by using the register name as a key.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

If a register requires more than one word to represent the data value the least significant word should be at reg_val, with more significant words at reg_val+1, etc. See Example 13-27.

Example 13-27. dsp_wreg

```
int devn;
int periph_num, reg_num;
unsigned long regval;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
regval=100L;
if (dsp_findreg(devn,"pc",&periph_num,&reg_num))
dsp_wreg(devn,periph_num,reg_num,&regval);
```

13.2.28 sim_docmd: Execute Simulator User Interface Command

```
sim_docmd(device_index,command_string)
int device_index;       /* DSP device index to be affected by command */
char *command_string;   /* User interface command to be executed */
```

This function executes any Simulator command that the Simulator normally accepts from the terminal. SIMDSP normally calls sim_gtcmd() to get a valid command string from the terminal, then calls sim_docmd to execute it. The device_index determines which dsp device (in a multiple dsp simulation) is affected by the command execution. The devices are numbered 0,1,2...n-1 in an n-device system, so be very careful, for example, to use 0 for the device_index parameter in a single device system.

If the command_string begins macro execution the selected device structure in_macro flag will be set by sim_docmd. SIMDSP retrieves valid commands from the macro file by calling sim_gmcmd() as long as the in_macro flag is set. The commands are still executed by sim_docmd, whether they come from the terminal or a macro file.

Commands which initiate device cycle execution (such as **go** or **trace**) will set the device structure sim_var.stat.executing flag. SIMDSP executes device cycles until the executing flag is cleared by an execution breakpoint. See Example 13-28

Example 13-28. sim_docmd

```
int devn;
dsp_startup();
devn=0;
dsp_new(devn,"56116");        /* Allocate structure for device 0, a 56116 */
sim_docmd(devn,"change pc $40");/* Change device 0 pc register to $40 */
sim_docmd(devn,"break r p:$80");/* Set a breakpoint for device 0 */
sim_docmd(devn,"go");         /* Begin execution of device 0 */
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.2.29 sim_gmcmd: Get Command String from Macro File

```
sim_gmcmd(device_index,command_string)
int device_index;       /* DSP device index to be affected by command */
char *command_string;   /* Pointer to return buffer for command line */
```

This function reads the next Simulator command string from a macro file. The `sim_docmd()` function will normally determine that a command is a macro, open the macro file and set the device structure `sim_const.in_macro` flag. The `sim_gmcmd()` function returns the next line from the open macro file each time it is called. It will clear the `in_macro` flag at the end of macro execution or if an invalid macro command is processed. The command_string buffer should be at least 80 characters. See Example 13-29.

Example 13-29. sim_gmcmd

```
/* Execute the macro command file startup.cmd on dsp device structure 0. */
#include "simcom.h"
#include "simusr.h"
extern struct sim_const sv_const;   /* Simulator device structures */
char command_string[80];
int devn;
dsp_startup();
devn=0;
dsp_new(devn,"56116");              /* Create new dsp structure */
sim_docmd(devn,"startup");          /* Begin the startup macro */
while (sv_const.in_macro){
sim_gmcmd(devn,command_string);     /* Get command string from macro file */
sim_docmd(devn,command_string);     /* Execute command string */
}
```

13.2.30 sim_gtcmd: Get Command String from Terminal

```
sim_gtcmd(device_index,command_string)
int device_index;       /* DSP device index to be affected by command */
char *command_string;   /* Pointer to return buffer for command line */
```

This function gets the next command string from the terminal in an interactive mode. The command line editing, command expansion and on-line help functions are invoked by this terminal command input function. The command string is fully checked for errors prior to returning. The command_string buffer should be at least 80 characters. See Example 13-30.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Example 13-30. sim_gtcmd

```
/* Get and execute Simulator commands for device 0 until a go type command is
*/
/* entered. */
#include "simcom.h"
#include "simusr.h"
extern struct sim_const sv_const;   /* Simulator device structures */
char command_string[80];
int devn;
dsp_startup();
devn=0;
dsp_new(devn,"56116");              /* Create new dsp structure */
while (!sv_const.sv[devn]->stat.executing){/* Check for go */
sim_gtcmd(devn,command_string);            /* Get command */
sim_docmd(devn,command_string);            /* Execute command */
}
```

13.3 Simulator External Memory Functions

The following sections describe functions which are provided in source code form in the Simulator package in the file simvmem.c. These functions define all the operations associated with reading, writing or storing dsp external memory locations. The Simulator memory allocation function is also included in this module since the representation of external memory is implemented with a virtual memory technique that is integrated with the memory allocation service. The external memory functions, with the exception of dsp_alloc, would not normally be called directly from the user's code. They are referenced from other Simulator functions, such as dsp_load or dsp_rmem, described in Section 13.2. Table 13-3 is a reference list of the external memory functions:

Table 13-3. External Memory Functions

Function	Description
dsp_alloc(num_bytes);	Allocate Simulator Program Memory
dspl_xmend(devn,map);	End DSP External Memory access
dspl_xmfree(devn);	Free DSP Device External Memory
dspl_xminit(devn);	Initialize DSP Device External Memory
dspl_xmload(devn,fp);	Load DSP External Memory from State File
dspl_xmnew(devn);	Create New External Memory Structure
dspl_xmrd(devn,map,add,val,fetch);	Read DSP External Memory Location to val

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 13-3. External Memory Functions (Continued)

Function	Description
`dspl_xmsave(devn,fp);`	Save DSP External Memory to State File
`dspl_xmstart(devn,map);`	Start DSP External Memory access
`dspl_xmwr(devn,map,add,val,store);`	Write DSP External Memory with val

The external memory access functions are provided in source form so that the external memory map attributes can be customized. This is especially useful for multiple dsp simulations in which complex configurations such as dual-port memory may be required. The functions in `simvmem.c` simulate the entire external memory space of up all dsp devices.

13.3.1 dsp_alloc: Allocate Simulator Program Memory

```
void *dsp_alloc(numbytes)
unsigned int numbytes;  /* Size of memory block needed in bytes */
```

This function must return a character pointer to the requested number of bytes of memory. It is not necessary for the memory to be cleared. A simple version could just call `malloc()`. The Simulator will not recover if the `dsp_alloc()` call fails, so an `exit()` must occur within `dsp_alloc()` if the requested memory cannot be allocated. See Example 13-31.

Example 13-31. dsp_alloc

```
/* Allocate memory for a new structure. */
void *dsp_alloc();
struct new_struct{
char buf[1000];
int bufindex;
} *newp;
newp=(struct new_struct *) dsp_alloc(sizeof(struct new_struct));
```

13.3.2 dspl_xmend: End DSP External Memory Access

```
dspl_xmend(device_index,memory_map)
int device_index;              /* DSP device index to be affected by command */
enum memory_map memory_map;    /* memory designator */
```

The core simulation calls this function during the last clock cycle of each external memory access. The `memory_map` parameter will be a memory designator as returned by `dsp_findmem`. Use the Simulator **help mem** command for a list of valid memory names. The `memory_map` enum is `memory_map_` concatenated with a valid memory name. As an example, `memory_map_pa` refers to off chip pa memory on the 96002 device.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.3.3 dspl_xmfree: Free DSP Device External Memory

```
dspl_xmfree(device_index)
int device_index;       /* DSP device index to be affected by command */
```

This function must free any memory that has been allocated to represent the external memory space of a selected device. Note that this function should not be called directly by the user's code. It is called as one of the steps in freeing an entire device structure by dsp_free(). See Example 13-32.

Example 13-32. dspl_xmfree

```
/* Free external memory of dsp device structure 0. */
int devn;
devn=0;
dspl_xmfree(devn);
```

13.3.4 dspl_xminit: Initialize DSP Device External Memory

```
dspl_xminit(device_index)
int device_index;       /* DSP device index to be affected by command */
```

This function must initialize the values in any structures used to represent the external memory of a dsp device. The Simulator commands **reset s** and **load s** will call dspl_xminit() in the processes of initializing or reloading the Simulator state. See Example 13-33.

Example 13-33. dspl_xminit

```
/* Initializing external memory for device 1 */
int devn;
devn=1;
dspl_xminit(devn);
```

13.3.5 dspl_xmload: Load DSP External Memory from State File

```
int dspl_xmload(device_index,fp)
int device_index;       /* DSP device index to be affected by command */
FILE *fp;               /* Pointer to file opened in text read mode ("r") */
```

This function must restore external memory from a Simulator state file. Note that this function should not be called directly by the user's code. The dspl_xmload() call is the last step of the dsp_load() function which loads a Simulator state file. The file pointer provided to dspl_xmload will have been opened with fp=fopen(filename,"r") and the remainder of the Simulator state will have already been restored from the state file. The steps used to restore the external memory should complement the steps used to save external memory in the dspl_xmsave() function. The return value of dspl_xmload() should be 1 if successful, 0 if an error occurred. The dsp_load() function will close the file following the dspl_xmload() call. See Example 13-34.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Example 13-34. dspl_xmload

```
/* Call of dspl_xmload() from dsp_load() */
int status;
FILE *fp;
fp=fopen(filename,"r");
./* Loading of other Simulator state structures */
.
status=dspl_xmload(devn,fp);
```

13.3.6 dspl_xmnew: Create New External Memory Structure

```
dspl_xmnew(device_index)
int device_index;        /* DSP device index to be affected by command */
```

This function must create and initialize the external memory for a device. Note that this function should not be called directly by the user's code. The dsp_new() function calls dspl_xmnew() as part of the process of creating a new dsp device structure. See Example 13-35

Example 13-35. dspl_xmnew

```
/* Call to dspl_xmnew() from dsp_new() */
dspl_xmnew(devn);
```

13.3.7 dspl_xmrd: Read DSP External Memory Location

```
int dspl_xmrd(device_index,mem_map,address,return_value,fetch)
int device_index;              /* DSP device index to be affected by command */
enum memory_map memory_map;    /* memory designator */
unsigned long address;         /* DSP memory address to read */
unsigned long *return_value;   /* Memory value will be returned here */
int fetch;                     /* Flag indicating that a dsp fetch is in
progress */
```

This function must return the value of a dsp device's external memory location. The Simulator calls dspl_xmrd() when a dsp device reads an external memory location, or when the Simulator user interface reads the location for display purposes. This function also returns a flag value of 1 if the memory location exists, 0 if it doesn't exist. The fetch parameter indicates to dspl_xmrd() whether or not the read is being executed by the dsp device. If fetch=1, the dsp device is fetching the memory location during execution of a device cycle. If fetch=0, dspl_xmrd() is being called from some other source not associated with device cycle execution (for example, from the memory display routines). Although the fetch parameter is not used in the version of dspl_xmrd() provided in the file simvmem.c, it is provided to enable special processing that should only occur when the device cycle simulation is taking place. The memory_map parameter will be a value representing the memory space being accessed. Use the Simulator **help mem** command for a list of valid memory names. The memory_map enum is memory_map_ concatenated with

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device. See Example 13-36.

Example 13-36. dsp_xmrd

```
/* Read "pe" memory location $5000 of device 2 */

unsigned long address;
int devindex;
unsigned long retval;
int ok;
int fetch;
address=0x5000L;
devindex=2;
fetch=0;
ok= dspl_xmrd(devindex,memory_map_pe,address,&retval,fetch);
```

13.3.8 dspl_xmsave: Save DSP External Memory to State File

```
int dspl_xmsave(device_index,fp)
int device_index;       /* DSP device index to be affected by command */
FILE *fp;               /* Pointer to file opened in write mode */
```

This function must save the external memory state to a Simulator state file. The dspl_xmsave() call is the last step of the dsp_save() function which saves a Simulator state file. The file pointer provided to dspl_xmsave will have been opened with fp=fopen(filename,"w+") and the remainder of the Simulator state will have already been saved to the state file. The return value of dspl_xmsave() should be 1 if successful, 0 if an error occurred. The dsp_save() function will close the file following the dspl_xmsave() call. See Example 13-37.

Example 13-37. dspl_xmsave

```
/* Call of dspl_xmsave() from dsp_save() */
int status;
FILE *fp;
fp=fopen(filename,"w+");
./* Saving of other Simulator state structures */
.
status=dspl_xmsave(devindex,fp);
```

13.3.9 dspl_xmstart: Start DSP External Memory Access

```
dspl_xmstart(device_index,memory_map)
int device_index;             /* DSP device index to be affected by command */
enum memory_map memory_map;   /* memory designator */
```

The core simulation calls this function during the first clock cycle of each external memory access. The memory_map parameter will be a value as returned by dsp_findmem. Use the Simulator **help mem** command for a list of valid memory names. The memory_map

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

enum is memory_map_ concatenated with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

13.3.10 dspl_xmwr: Write DSP External Memory Location

```
int dspl_xmwr(device_index,mem_map,address,value,store)
int device_index;             /* DSP device index to be affected by command */
enum memory_map memory_map;   /* memory designator */
unsigned long address;        /* DSP memory address to write */
unsigned long value;          /* Value to be written to memory location */
int store;                    /* Flag indicating that a device store is in
effect */
```

This function must store a value to a dsp device's external memory location. The Simulator calls dspl_xmwr() when a dsp device writes an external memory location, or when the Simulator user interface alters the location. The store parameter will indicate if the reference is from the dsp device (store=1) during simulation of device cycle execution, or some other source (store=0) not related to device cycle execution. For example, the **change** memory Simulator command will set the parameter store to 0. The store parameter is not used in the dspl_xmwr function provided in the file simvmem.c, but is available to the user if modifications are made to the simvmem.c file for special external memory processing. The memory_map parameter will be a value representing the memory space being accessed. Use the Simulator's **help mem** command to obtain a list of the valid memory space prefixes. See Example 13-38.

Example 13-38. dspl_xmwr

```
/* Write value of 3 to "xe" memory location 5 of device 2 */

unsigned long address;
int devindex;
int ok;
unsigned long newval;
int store;
address=5L;
devindex=2;
newval=3L;
store=0;
ok=     dspl_xmwr(devindex,memory_map_xe,address,newval,store);
```

13.4 Simulator Screen Management Functions

The following sections describe functions which are provided in source code form in the Simulator package in the file scrmgr.c. These functions define all the operations associated with Simulator terminal I/O. The code includes conditionally compiled sections for MSDOS, UNIX, and VMS. The code is provided to allow customization of the Simulator terminal I/O for a particular environment. The user may, for example, wish to

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

redefine the control characters used by the Simulator so that they map to some particular terminal.

Table 13-4 is a quick reference list of the Simulator screen management functions:

Table 13-4. Screen Management Functions

Function	Description
simw_ceol();	Clear to end of line
simw_ctrlbr();	Check for CTRL-C signal
simw_cursor(line,column);	Move cursor to specified line, column
simw_endwin();	End the Simulator display
simw_getch();	Non-translated keyboard input
simw_gkey();	Translated keyboard input
simw_putc(c);	Output character to terminal
simw_puts(line,column,text,flag);	Output string to terminal at line and column
simw_redo(device);	Repaint screen with output from device
simw_redraw(count);	Redraw screen after scrolling count
simw_refresh();	Screen update after buffering output
simw_scrnest();	Nest output buffering another level
simw_unnest();	Pop output buffering one level
simw_winit();	Initialize window parameters
simw_wscr(string,commandflag);	Write string and perform logging functions

13.4.1 simw_ceol: Clear to End of Line

simw_ceol()

This function must clear the display from the current column to the end of line, then return the cursor to the previous position.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.4.2 simw_ctrlbr: Check for CTRL-C Signal

`simw_ctrlbr()`

This function must check for the occurrence of a CTRL-C signal from the terminal. If the CTRL-C signal occurs, it sets a flag for the active breakpoint dsp (defined by `sv_const.breakdev`). It returns the `sim_var.stat.CTRLBR` flag for the current device. This allows the program to select the device that will halt in response to the CTRL-C signal from the keyboard in a multiple device simulation.

13.4.3 simw_cursor: Move Cursor to Specified Line and Column

`simw_cursor(line,column)`

This function must move the cursor to the specified line and column and update the `sim_const.curline` and `sim_const.curclm` variables.

13.4.4 simw_endwin: End Simulator Window

`simw_endwin()`

This function is normally called when returning to the operating system level from the Simulator. It must terminate any special processing associated with terminal I/O for the Simulator and clear the display.

13.4.5 simw_getch: Non-translated Keyboard Input

`simw_getch()`

This function gets a single character in a non-translated mode from the terminal. It is not used much by the Simulator - only when returning from the execution of the **system** command prior to the time when the Simulator's special terminal I/O processing is reinitialized.

13.4.6 simw_gkey: Translated Keyboard Input

`simw_gkey()`

This function gets a keystroke from the terminal and maps it to one of the accepted internal codes used by the Simulator. The internal codes are defined in `simusr.h`. This function should not output the character to the terminal. This function is a good candidate for modification if you want to change the set of input control characters used by the Simulator.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

13.4.7 simw_putc: Output Character to Terminal

```
simw_putc(c)
char c;
```

This function outputs the character in the variable `c` at the current cursor and column position. It advances and updates the `sim_const.curclm` variable. This function is not used often by the Simulator, and it is not very time critical when it is used, so the Simulator implementation is just to call `simw_puts()` after creating a temporary string from the character `c`.

13.4.8 simw_puts: Output String to Terminal

```
simw_puts(line,column,text,flag)
int line;               /* Move cursor to this line for output */
int column;             /* Move cursor to this column for output */
char *text;             /* Text string to be output */
int flag;               /* 0=non-bold, 1=bold on/off by {}, 2=all bold */
```

This function outputs a string to the terminal at the specified line and column. Highlighting of output can be enabled either by setting the `flag` parameter to 2 or by enclosing text in curly braces and setting the `flag` parameter to 1.

13.4.9 simw_redo: Repaint Screen With Output From Device

```
simw_redo(device)
int device;             /* Use screen buffer from this device to repaint
screen */
```

This function repaints the screen from a device screen buffer. It is normally only called when re-entering the Simulator following a **system** command, after loading the device state with the **load s** *filename* command, or after switching devices in a multiple device simulation with the **device** command.

13.4.10 simw_redraw: Redraw Screen After Scroll Count

```
simw_redraw(count)
int count;              /* Number of lines to scroll before repainting the
screen */
```

This function scrolls up or down `count` lines in the display buffer, then redisplays the text in the buffer at that position. This function only displays the text that is in the scrolling portion of the display.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

13.4.11 simw_refresh: Screen Update After Buffering Output

`simw_refresh()`

The Simulator buffers screen output in implementations other than MSDOS in order to decrease the time spent repainting the screen. This provides a fixed display effect for consecutive **trace** commands. The `simw_refresh()` function will take care of refreshing the screen following buffering of screen output. It also resets the `sim_const.scrnest` variable to 0 to coincide with the non-buffered status of the screen following the refresh.

13.4.12 simw_scrnest: Increase Screen Buffering One Level

`simw_scrnest()`

This function increments a counter to signify the screen output buffering level. The companion `simw_unnest()` and `simw_refresh()` functions provide the output buffering operations for the Simulator. The `sim_const.scrnest` variable is incremented each time this function is called.

13.4.13 simw_unnest: Decrease Screen Buffering One Level

`simw_unnest()`

This function decrements the `sim_const.scrnest` variable each time it is called. If the screen buffering level drops below one, `simw_unnest()` will call `simw_refresh()` to update the screen.

13.4.14 simw_winit: Initialize Window Parameters

`simw_winit()`

This function initializes any screen or keyboard parameters that are required for the Simulator terminal I/O environment. It is called whenever the Simulator is entered from the operating system level, which includes the initial Simulator entry and re- entry following the **system** command.

13.4.15 simw_wscr: Write String and Perform Logging

```
simw_wscr(text,command_flag)
char *text;             /* Text string to write to screen */
int command_flag;       /* Flag 1=string is a command, 0= not a command */
```

This function outputs the string to the terminal above the command line after scrolling the display up one line. It also takes care of writing the text string to the proper log files specified by the Simulator **log s** or **log c** commands.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.5 Non-Display Simulator

The Simulator package contains object libraries which support both display and non-display versions of the Simulator. The library **nwsim** contains functions available to the non-display version of the Simulator. The library **wwsim** contains functions that may only be used in a display version of the Simulator. For each device type there are also display and non-display device-specific libraries named **wwxxxxx** and **nwxxxxx** where the **xxxxx** is the device number.

The source code contained in `snwdsp.c` can be linked with the nwxxxxx and nwsim libraries to create a non-display version of the Simulator. Elimination of the user interface functions cuts the code size of the Simulator almost in half. However, all of the functions listed in Section 13.4 and `sim_docmd()`, `sim_gmcmd()` and `sim_gtcmd()` described in Section 13.2, are sacrificed.

The remainder of the functions in Section 13.2 and all of the functions in Section 13.3 are available in the non-display Simulator libraries.

Some major features of the Simulator are eliminated by the loss of the `sim_docmd()` function. In particular, there are no low-level entry points provided to set a breakpoint or to assign input or output files to DSP peripheral functions. However, the basic functions required to create a device, load a program, execute the code, and test or modify device registers are all still available. In addition, the `dsp_save()` function provides the capability to save the state of the non-display version. The state file can later be reloaded by a display version of the Simulator for visual examination of the registers and memory contents.

The following sections cover several topics that concern the non-display version of the Simulator. Section 13.5.1 deals with creating a new device. Section 13.5.2 describes how to load a program or state file. Section 13.5.3 describes how to execute device cycles. Section 13.5.4 describes how to test breakpoint conditions.

13.5.1 Creating a New Device

The `simcom.h` file defines the maximum number of DSP devices in the constant `DSP_MAXDEVICES`. A new device can be created and numbered from 0 to `DSP_MAXDEVICES-1`. The structures are allocated by calls to the `dsp_new()` function described in Section 13.2.13, "dsp_new: Create New DSP Device Structure," . See Example 13-39

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Example 13-39. Code to Create New Device

The following C source code illustrates the steps necessary to create 3 DSP devices.

```
dsp_startup();
dsp_new(0,"56116");          /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");          /* Allocate structure for device 1, a 56116 */
dsp_new(2,"56116");          /* Allocate structure for device 2, a 56116 */
```

13.5.2 Loading Program Code or Device State

The display version of the Simulator provides the high level `sim_docmd()` function interface. It allows the user to simply execute the high level **load** or **load s** Simulator commands to load program code or a Simulator state file. The non-display version of the Simulator makes use of the lower level function calls, `dsp_ldmem()` and `dsp_load()`, to accomplish the same results. They are described in Section 13.2. The major difference from their high-level counterparts is that no file-name expansion is provided in the lower level calls.

The program code loaded by the `dsp_ldmem()` function may be any COFF format or OMF format file. The OMF format is created as the output of versions of the macro assembler prior to release 4.0 and of the Simulator **save** command.The COFF format files are the output of the macro assembler beginning with release 4.0, or those saved by the Simulator **save** command with the suffix ".cld".

The Simulator state loaded by the `dsp_load()` function may have previously been saved by a display or non-display version of the Simulator. The formats are the same.

The `dsp_save()` function is provided as a low-level entry point that saves the Simulator state for a non-display version of the Simulator. It is the same function that is called during execution of the high level **save s** command, which is only available in the display version. The only limitation is that the full save filename must be specified. No automatic expansion is done for the working path or filename suffix as in the higher level Simulator calls. The `dsp_save()` function is described in Section 13.2, "Simulator Object Library Entry Points," .

13.5.3 Executing Device Cycles

After creating a new device - as described in Section 13.5.1, "Creating a New Device," - and loading a program or state file - as described in Section 13.5.2, "Loading Program Code or Device State," - the Simulator is ready to execute the program code.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Execution will begin at the start address specified in the load file, or continue from the previous location in a Simulator state file. The user's code may select a new execution address by writing register "pc" using the dsp_wreg function.

The Simulator will advance the device state by one clock cycle each time the dsp_exec function is called. The device pin states are updated each clock cycle, and can be examined or changed using the dsp_rpin, dsp_wpin, dsp_rport, or dsp_wport functions.

13.5.4 Testing Breakpoint Conditions

The display version of the Simulator provides a way to specify breakpoint conditions that are evaluated each time dsp_exec is called. If the breakpoint condition is met, the Simulator displays the enabled registers and clears the device structure sim_var.stat.executing flag (assuming the breakpoint action is halt).

The non-display Simulator does not provide a way to specify breakpoint conditions. It is up to the user's code to examine device registers or memory conditions and decide whether or not to continue cycle execution. The device registers and memory can be examined using the dsp_rreg and dsp_rmem functions. The example program snwdsp.c simply checks the Simulator cycle counter for device 0 and terminates execution after some number of cycles.

Another variable that may be particularly useful in breakpoint testing is dev_var.flg_stat. It maintains bit flags which signal end-of-instruction (DSP_GEOI), end of repeat cycle (DSP_GEOR), and illegal opcode (DSP_GILLEG). The bit flag definitions are defined in simdev.h.

13.6 Multiple Device Simulation

The SIMDSP Simulator initially simulates a single dsp device; but additional devices can be created using the **device** command. Device to device pin connections or device to device memory map connections can be specified with the Simulator **input** command. The following sections describe some details about the way the Simulator handles multiple device simulation. Section 13.6.1 describes the required steps which allocate and initialize multiple dsp structures. Section 13.6.2 describes the method of interleaving device execution in order to maintain multiple device synchronization. Section 13.6.3 describes simulation of the external memory space of the dsp devices. Section 13.6.4 describes multiple device pin connections. Section 13.6.5 describes display of device output in the multiple device environment.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

13.6.1 Allocation and Initialization of Multiple Devices

Most of the higher level Simulator functions require a device index as one of the parameters. The Simulator uses the device index to select a previously allocated DSP structure. The DSP structures are allocated dynamically by calling the `dsp_new` function for each device. The device type is also selected in the `dsp_new` function call. In the display version of the Simulator, the **device** command handles the details of calling `dsp_new`. The proper sequence of instructions necessary to allocate three DSP devices is shown below.

```
dsp_startup();
dsp_new(0,"56116");          /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");          /* Allocate structure for device 1, a 56116 */
dsp_new(2,"56116");          /* Allocate structure for device 2, a 56116 */
```

13.6.2 Interleaving Multiple DSP Simulations

The `dsp_exec` function executes a single DSP clock cycle and updates the selected DSP device structure. In order to simulate simultaneous multiple DSP execution, `dsp_exec` should be called for every device before proceeding to the next clock cycle. The `simdsp` Simulator executes a single clock cycle for each active dsp device, then halts if any active device has cleared its `sim_var.stat`.executing flag. It allows Simulator commands, such as register or memory modifications, on the viewed device until a command sets the executing flag again. The device which causes a breakpoint becomes the viewed device by default, but the viewed device can be changed with the **device** command without changing the status of any device. A particular device can be halted by setting the `CTRLBR` flag in its `sim_var.stat` structure. This has the same effect as typing CTRL-C at the keyboard while a device is running. It breaks device execution at the end of the current instruction. Note that it is not mandatory to wait for the `sim_var.stat`.executing flag to be set to begin device execution, or to halt if the executing flag is clear. These are just convenience features for the Simulator user interface. Device cycle execution can be advanced in single cycle increments at any time by calling `dsp_exec`.

13.6.3 External Memory Definition

The Simulator package contains the C source file `simvmem.c` which contains all the external memory access functions used by the SIMDSP Simulator. These functions are described in detail in Section 13.3. The functions, as written, will automatically simulate the entire external memory space of all DSP devices, assuming that the operating system can allocate enough memory to store the device structures for the devices.

The user may wish to modify the `simvmem.c` functions in order to define special external memory configurations. The functions can be modified, for example, to simulate the

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

response of dual-port RAM or special memory-mapped peripherals. Another good reason to modify the external memory functions is to increase the speed of the simulation. If the user's simulation only requires some minimum amount of external memory, then the virtual memory management functions provided with the Simulator are probably overkill.

13.6.4 Multiple DSP Pin Interconnections

The `dsp_exec` function will automatically update the DSP device pin states by one clock cycle change each time it is called. The display version of the Simulator will also retrieve or send data to assigned I/O files as defined by the **input** and **output** commands. The **input** command supplies a method of connecting device pins back to other device pins on the same device as well as to device pins on another device.

The device pin states for any device can be examined or written using the `dsp_rpin` and `dsp_wpin` functions described in Section 13.2. Simulation of pin to pin connection simply requires reading the state of the output pin each cycle with `dsp_rpin` and writing it to the input pin with `dsp_wpin`. A bidirectional pin connection requires reading and writing both pins. The Simulator maintains separate buffers for input and output data for each pin, so there is no problem writing a pin, even if it is defined as an output. The input value will be stored, but will only be used if the pin is subsequently reconfigured as an input.

An entire port state can be read or written using the `dsp_rport` and `dsp_wport` functions described in Section 13.2. The port and pin states are derived from the same storage variables in the device structure. The `dsp_rport`, `dsp_wport`, `dsp_rpin`, and `dsp_wpin` functions just provide a convenient method of retrieving the data from this structure.

13.6.5 Multiple DSP Simulator Display

The Simulator display functions are contained in the source file `scrmgr.c` in the Simulator package. This code supports a virtual screen for each simulated dsp.

The supplied display code uses a single window. The lines above the command line form a scrollable region in which session output is displayed. The command line, error line and help line are the three bottom lines of the display. Each allocated device contains screen buffer memory which saves the previous 100 lines of output which is written to a device's scroll region. The terminal screen update is inhibited unless the `sim_const.viewdev` value matches the device index, but the output is always placed in the device's screen buffer. The screen can be completely refreshed from a selected device screen buffer by executing the `simw_redo` function.

The **device** command allows the user to switch the displayed device. When it switches to a new device, it refreshes the entire screen from the device's display buffer.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

13.7 Reserved Function Names

The public function names used in the Simulator all begin with the prefixes dsp or sim. Functions which begin with sim are only available when a display version of the Simulator is created. Functions which begin with dsp are available to both display and non display versions. The screen management functions all begin with simw_. In general, functions which begin with dsp_ or sim_ are higher level functions available for direct reference from the user's code; those beginning with dspl_ or siml_ are meant only for internal use by the Simulator. The higher level functions and the screen management functions are documented in Sections 13.2, 13.3, and 13.4. The public function names are listed in the file named global.sym which is included with the distribution.

13.8 Simulator Global Variables

In order to reduce conflicts with user variable names, the Simulator global variables have been grouped together into several large structures. In general, the structure names beginning with s are used defined in simusr.h and are only used in the display version of the Simulator; while those beginning with d are defined in simdev.h and are used by both the display and non-display versions of the Simulator. The prefixes st_ and dt_ are used for structure names of device-type structures, that is structures which must be defined for each device type. The prefixes sim_ and dev_ are used for structure names of general device or simulation structures.

Global variable names may have a prefix dx_, dv_, sx_, or sv_. The prefix dx_ is used for variables of dt_ structures. The prefix dv_ is used for variables of dev_ structures. The prefix sx_ is used for variables of st_ structures. The prefix sv_ is used for variables of sev_ structures. A list of Simulator global variables is included in the distribution file named global.sym.

13.9 Modification of Simulator Global Structures

The source file simglob.c, which is included in the Simulator package, contains the global structures sv_const and dv_const. There are some useful modifications, described below, that can be made to the constant definitions at the beginning of simglob.c. The simglob.c module must then be recompiled and relinked using the make file provided with the Simulator package. In addition to these, there may be device-specific modifications that can be made to the Simulator.

DSP_MAXDEVICES This define constant determines the maximum number of devices that can be allocated using the Simulator's **device** command. As a default it is set to 32.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

DSP_CMDSZ This define constant determines the size of the previous command stack. The Simulator commands are stored in the stack and can be reviewed using the **ctrl-f** and **ctrl-b** key entries. As a default the previous command stack size is set to 10.

DSP_HISTSZ This define constant determines the size of the execution history buffer, which stores device instructions as the Simulator executes. The buffer is used by the Simulator **history** command, and has a default size that can save 32 instruction words.

DSP_WINSZ This define constant determines the size of the screen buffer that is maintained and displayed by the `scrmgr.c` functions. It specifies the number of display lines that will be allocated for each device as they are created with the Simulator **device** command. The user can use the **ctrl-u**, **ctrl-t**, **ctrl-v**, and **ctrl-d** key sequences to review display lines that have scrolled off the screen. This constant should not be set to a value smaller than the number of lines in the display window.

13.10 Modification of Device Global Structures

The device types available to the simulation are defined in the source module named with the prefix `dsp` followed by the device family name. As an example, the file `dsp56100.c` contains the global structures `dx_56116` and `sx_56116`, which define device-specific information about the DSP56116 device in the 56100 family. You may wish to modify this module to define a new device type that can then be created by the simulator's **device** command. The basic idea is to create a new device type and definition by modifying a previous definition in the `dsp` file. As an example, using the 56116 device in the file `dsp56100.c`, the procedure would be:

Make a copy of `dsp56100.c` and name it something else. Modify the `makefile` file to include this new module name for compilation and linking in the same manner that `makefile` handles the dsp56116.

In the new file, rename the `dx_56116` structure to some name other than `dx_56116`, and put a pointer to this new structure in the `dx_all` array in the file `simglob.c`.

In the new `dt_var` structure, change the device type name to some name other than 56116 - this is in the first member of the `dt_var` structure

In the new file, change the `sx_56116` structure to some name other than `sx_56116`, and put a pointer to this new structure in the `sx_all` array in the file `simglob.c`.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

After the above steps are completed, lower level structures and define constants in the new module can be modified to change such parameters as on-chip memory size, number of peripherals, or number and names of pins. Continuing with the 56116 example, Table 13-5 is a list of the lower level structures and define constants associated with the 56116 that may be changed in the new file:

Table 13-5. Lower Level Structures & Define Constants

Function	Description
DSP_PI_SIZE_116	This define constant determines the size of the on-chip program memory.
DSP_PR_SIZE_116	This define constant determines the size of the on-chip bootstrap rom memory.
DSP_XI_SIZE_116	This define constant determines the size of the on-chip X data memory.
DSP_XP_SIZE_116	This define constant determines the size of the on-chip X memory-mapped peripheral register space.
dx_periph_56116	This structure can be modified to add peripherals to or remove peripherals from the newly defined device. If you modify this structure, you must also modify the sx_periph_56116 structure in a similar manner. The file portb100.c is provided in source form with the simulator package as a model to be used when creating new peripheral structures.
bootrom	This is the default initialization data for the on-chip bootstrap rom. It is loaded at start-up and in response to the **reset s** command. You can also change the contents of the bootstrap rom with the simulator **asm**, **load**, and **change** commands.
xpin	This structure determines the names assigned to device pins, as well as the order in which the pins are displayed in output pin lists. It also provides a cross-reference from the pin name to the physical bit and storage port in which the simulator maintains the pin data. The portindex cross-reference is an offset to the proper dev_xpval structure from the xportval pointer in the dev_var structure. Each pin has a primary name and a possible alternate peripheral function pin name. You may modify the names or delete or add dt_xpin structures from this array, but do not change the portindex and pinmask members of the dt_xpin structures.
mem_56116	This array of dt_memory structures can be modified to change parameters associated with the DSP56116 memory attributes. The name member is the memory name used by the simulator commands to reference the memory space. The memsize member determines the size of arrays allocated for on-chip memory. The memattr member determines whether the memory is located on or off chip and whether it is ram or rom. The romval pointer can point to initialization data for the memory space. If it is NULL, the memory space will be initialized to zero at start-up and in response to the **reset s** command. Although you may change the memory names, memory size, memory attributes, and initialization data, do not add or delete dt_memory structures from the mem_56116 array.
pval	This array of dt_xpidata structures is used by the simulator to initialize the input pin data in the dev_var.xportval structures at simulator start-up and in response to the **reset s** command.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Table 13-5. Lower Level Structures & Define Constants

Function	Description
xports	This array of dt_xpid structures determines port names that can be used in the simulator input and output commands. These names are in addition to the peripheral names that are specified in the individual peripheral modules. The port names are just a convenient way to specify a subgroup of pins within a single physical port for input and output operations.
dx_56116	This structure is mostly a conglomeration of the other substructures defined within the module for this device type. The only member of this structure that should be modified is devname, which will be used to specify the new device type in the simulator **device** command
mem_dispfw56116	This structure provides display field width information for the different memory spaces defined in the mem_56116 defined previously.
hlp_56116	This structure provides the help pointers that will be used by the simulator when help is requested for this device type.
sx_periph_56116	This structure points to display information for the registers of each peripheral; the actual display information is defined locally in each peripheral module. The file portb100.c is provided in source form with the simulator package as a model to be used when creating new peripheral structures.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Index

A

alternate directory 3-1
ASM
    examples 12-7, 12-8
ASM - Single Line Interactive Assembler 12-7
Assembly 11-5, 11-6
assign 5-11
    output file 5-11, 5-12, 5-13

B

binary 6-3
break 2-4, 2-7, 12-8
breakpoint 1-15, 12-12
    action 12-10
    clear 13-9
    examples 12-12
    testing 13-6, 13-19, 13-33
Breakpoints 1-13, 11-1, 11-5, 11-6
byte wide 12-7

C

C 7-1
Call Stack 7-1, 7-2
Calls 7-4
CHANGE 13-26
change 12-13
    examples 12-14
.cld 13-10
.cmd 13-20
COFF 3-3
command 8-1, 8-2, 11-2, 11-3, 12-5, 12-6
    ASM 12-7
    DEVICE 12-15
    DISASSEMBLE 12-16
    DISPLAY 12-18
    DOWN 12-19
    entering 1-2
    FINISH 12-21
    HISTORY 12-24
    INPUT 12-24, 12-25
    LIST 12-27
command entry
    command line editing 13-20
    expansion 13-20
    from macro file 13-19, 13-20
    from terminal 13-19, 13-20, 13-28
command execution
    go mode 13-19
    macros 13-19, 13-20
    trace mode 13-19, 13-30
Command Overview 12-1
Command Syntax 12-4
commands
    load 13-32
    system 13-28, 13-29
comment
    from dspt_masm 13-4
configuration
    custom external memory 13-22, 13-34
Copy 12-14
    examples 12-14
Copy Memory 4-6
create 8-1
CTRLBR 13-34
CTRL-C 13-28, 13-34

D

data 5-9, 5-10
debugging 7-1
decimal 6-3
default 6-2
Device 5-11, 5-12, 6-2, 11-3
Device Button 10-6
Device IO 5-1, 6-2
    introduction 5-1
device type 12-15
directory 3-1, 3-2
disable 11-1, 11-6
Disassemble 4-7, 11-5
Display 6-6, 11-1, 11-2, 11-3, 11-5
Display menu 3-1
    Path 3-1, 3-2
Down 7-2, 7-3, 12-19
down
    examples 12-19
dsp_alloc 13-22
dsp_exec 13-6
dsp_findmem 13-6
dsp_findpin 13-7
dsp_findport 13-7
dsp_findreg 13-8
dsp_fmem 13-9

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

dsp_free 13-8
DSP_GEOI 13-33
DSP_GEOR 13-33
DSP_GILLEG 13-33
dsp_init 13-10
dsp_ldmem 13-10, 13-32
dsp_load 13-11, 13-32
dsp_new 13-11, 13-16, 13-31
dsp_path 13-11
dsp_rapin 13-12
dsp_rmem 13-13
dsp_rpin 13-13, 13-35
dsp_rport 13-14, 13-35
dsp_rreg 13-15
dsp_save 13-15, 13-32
dsp_startup 13-15
dsp_unlock 13-16
dsp_wapin 13-16
dsp_wmem 13-17
dsp_wpin 13-17, 13-35
dsp_wport 13-18, 13-35
dsp_wreg 13-19
dspl_xmend 13-22
dspl_xmfree 13-23
dspl_xminit 13-23
dspl_xmload 13-23
dspl_xmnew 13-24
dspl_xmrd 13-24
dspl_xmsave 13-25
dspl_xmstart 13-25
dspl_xmwr 13-26
dspt_masm_xxxxx 13-5
dspt_unasm_xxxxx 13-5
during 2-6

E

edit 11-5, 11-6
enable 11-1, 11-5
entering commands 1-2, 11-2
Evaluate 9-1, 9-2
execute 2-7
    GO command 12-22
    pause 2-6
Execute menu 2-7
    Finish 2-6
    GO 1-13, 1-15, 2-1
    Reset 2-7, 2-8
    Stop 2-7
executing 13-6, 13-19, 13-33
Exit 6-6
expression 12-20
    EVALUATE 12-20
expressions 9-1
    introduction 9-1

F

Features 1-2
file i/o
    comands 13-20
    commands 13-19
    filename 13-11
    initialization 13-9
    memory 13-32
    object module 13-10
    state file 13-10, 13-15, 13-23, 13-25, 13-32
File menu 3-3
    Load 6-4, 6-5
    Macro 8-1, 8-2
    Save 3-3
filename suffixes
    .cld 13-10
    .lod 13-10
    .sim 13-10, 13-15
files 6-4
FINISH 2-6
Finish 2-6
FINISH Button 10-5
fonts 6-6
format 5-1
    I/O files 5-1
Frame 7-2, 7-3
frame 12-21
    examples 12-21
function 2-6

G

Getting Started 1-3
global variables 13-36
Go 12-22

H

Help 12-23
help 11-3
hexadecimal 6-3
History 4-8, 11-2

I

I/O files 5-1
    formatting 5-1
in_macro 13-19, 13-20
Initialization
    of a particular device 13-9
    of external memory 13-23
initialization

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

by dsp_new 13-11
dsp_init 13-9
of external memory 13-24
of window parameters 13-30
Input 5-8, 5-9, 5-10, 12-24, 12-25
examples 12-26
instruction 2-1, 2-2, 11-5, 11-6
interrupt 2-7
Introduction to the DSP Simulator 1-1
IO Redirect 7-5
IO Streams 7-5

L

line 1-2, 1-3, 2-2
List 11-1, 12-27
examples 12-27
Load 1-6, 6-4
.lod 13-10
Log 8-2
LOG - Log Commands
Profile 12-30
Session 12-30
Profile 12-31
Log Files 8-1

M

macro command file
execution 13-19, 13-20
macros 8-1
malloc 13-22
memory
access functions 13-1, 13-22, 13-24, 13-25, 13-26, 13-34
allocation 13-22
change values 12-13
change values in 4-5
display 4-3, 4-4
free 13-8, 13-23
initialization 13-23
initialize 13-9
load 13-10
load from state file 13-23
read 13-12
resetting 2-7
save to state file 13-25
write 13-9, 13-16
mnemonic 12-7
modify 4-3
memory 4-5
registers 4-3
Modify menu 6-3
Device 6-2
Radix 6-2
multiple dsp simulation
device index 13-19
halting 13-28
interleaving execution 13-34

N

Next 2-3, 11-5, 11-6
examples 12-32
next 12-32
Next Button 10-4, 10-5
non-display simulation
executing device cycles 13-33
library file 13-31
loading program code 13-32
testing breakpoint conditions 13-33
numbering system 6-2
nwsim 13-31
nwxxxxx 13-31

O

object code 1-6, 3-3, 11-5, 12-16
object module
loading 13-10
OMF 3-3
Output 5-11, 5-12, 5-13, 11-7, 12-33

P

parameters 12-5
Path 3-1, 12-35
examples 12-35
pause 2-7
peripheral 5-8, 5-9, 5-10
pins 5-10
preferences 6-4

Q

QUIT- Quit Simulator Session 12-36

R

Radix 6-3, 6-4, 11-2, 12-36
RADIX - Change Input or Display Radix 12-36
Redirect 12-38
examples 12-39
REDIRECT- Redirect stdin/stdout/stderr for C Programs 12-38
register 4-1, 4-2, 12-17, 12-18
change values 12-13
program counter 13-33
read function 13-14

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

write function 13-18
registers 2-7
change values of 4-3
resetting 2-7
REPEAT Button 10-7
Reset 2-7
examples 12-39
RESET Button 10-8
RESET- Reset Device or State 12-39
RTS 2-6
run 8-2

S

Save 3-3, 6-6, 12-40
examples 12-40
SAVE- Save Simulator File 12-40
screen buffer 13-30
scripts 8-1
scrmgr.c 13-26, 13-35
Session 11-4, 11-5
.sim 13-10, 13-15
sim_docmd 13-19
sim_gmcmd 13-20
sim_gtcmd 13-21
Simulator 1-1
Introduction 1-1
memory configuration 6-1
preferences 6-6
running 1-3
simvmem.c 13-22, 13-34
simw_ceol 13-27
simw_ctrlbr 13-28
simw_cursor 13-28
simw_endwin 13-28
simw_getch 13-28
simw_gkey 13-28
simw_putc 13-29
simw_puts 13-29
simw_redo 13-29, 13-35
simw_redraw 13-29
simw_refresh 13-30
simw_scrnest 13-30
simw_unnest 13-30
simw_winit 13-30
simw_wscr 13-30
Source 11-6
source code 1-6, 3-3, 7-1, 11-5
Stack 7-2, 7-3, 11-7
start execution 1-13
state 2-7, 2-8, 6-4, 6-5
resetting 2-7
state file
create 13-15, 13-25
load 13-23
Step 12-41
examples 12-41
step 12-32
STEP - Step Through DSP Program 12-41
STEP Button 10-3
stop 2-6, 2-7
STOP Button 10-2
Streams 12-42
examples 12-42
STREAMS - Enable/Disable Handling of I/O for C Programs 12-42
subroutine 2-6
syntax 12-4
of commands 12-4
System
examples 12-43
SYSTEM - Execute System Command 12-43

T

terminal
i/o functions 13-1
Trace 2-3, 12-44
examples 12-44
TRACE - Trace Through DSP Program 12-44
Type 7-6
examples 12-45
TYPE - Display the Result Type of C Expression 12-45

U

unlock 12-15
UNLOCK - Unlock Password Protected Device Type 12-46
Until 2-4
examples 12-46
UNTIL - Step Until Address 12-46
Up 7-2
UP - Move Up the C Function Call Stack 12-47
Using the Tool Bar 10-1

V

VIEW - Select Display Mode 12-47

W

Wait 12-48
WASM 12-48
Watch 9-8, 9-9
examples 12-49
WATCH - Set 12-49
Modify
View

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

or Clear Watch item 12-49
WBREAKPOINT - GUI Breakpoint Window 12-50
WCALLS - GUI C Calls Stack window 12-50
WCOMMAND - GUI Command window 12-51
WHERE - GUI C Calls Stack window 12-51
windows 6-6, 11-2, 11-5
WINPUT - GUI File Input window 12-52
WLIST - GUI list window 12-52
working directory 3-1, 3-2
WREGISTER - GUI Register window 12-54
WSESSION - GUI Session window 12-55
WSOURCE - GUI Source window 12-55
WSTACK - GUI Stack window 7-2, 12-56
wwatch
examples 12-57
WWATCH - GUI Watch window 12-56
wwsim 13-31
wwxxxxx 13-31

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com